Univerzita Karlova

Pedagogická fakulta

Katedra informačních technologií a technické výchovy

BAKALÁŘSKÁ PRÁCE

Materiály pro vzdělávání programování s využitím grafiky

Materials designated for programming education focusing on graphics

Emma Strouhalová

Vedoucí bakalářské práce:	RNDr. Michal Töpfer
Studijní program:	Specializace v pedagogice (B7507)
Studijní obor:	Informační technologie se zaměřením na vzdělávání (OKB1IT17)

Praha 2024

Odevzdáním této bakalářské práce na téma Materiály pro vzdělávání programování s využitím grafiky potvrzuji, že jsem ji vypracovala pod vedením vedoucího práce samostatně za použití v práci uvedených pramenů a literatury. Dále potvrzuji, že tato práce nebyla využita k získání jiného nebo stejného titulu.

Praha, 2. prosince 2024

Na tomto místě bych velice ráda poděkovala vedoucímu mé bakalářské práce, RNDr. Michalu Töpferovi, za konzultace a veškerý čas, který mojí práci obětavě věnoval, odborné vedení, užitečné připomínky a cenné rady, které mi při psaní práce poskytl.

Abstrakt

Tato bakalářská práce se zaměřuje na vytvoření vzdělávacích materiálů pro výuku programování s využitím počítačové grafiky. Cílem práce je nabídnout studentům sadu praktických úloh, na kterých si mohou procvičit své programátorské dovednosti. Úlohy jsou gradované a zahrnují nejen potřebné teoretické vysvětlení, ale i návod, který studentům umožňuje samostatnou práci, což je činí ideálními pro procvičování. Automatické generátory vstupních dat a kontrolní programy navíc studentům poskytují okamžitou zpětnou vazbu k jejich řešením. Studenti také získají podporu v knihovně s pomocnými funkcemi. Úlohy jsou navrženy tak, aby byly univerzální a nezávislé na konkrétním programovacím jazyku. Praktické využití materiálů je ukázáno na příkladech z oblasti grafiky, jako je vykreslování čar, mnohoúhelníků a anti-aliasing.

Klíčová slova

výuka programování, počítačová grafika, vzdělávací materiály, vzdělávací úlohy, gradované úlohy, kreslení čar, grafické algoritmy, open-datové úlohy, Kotlin, anti-aliasing

Abstract

This bachelor's thesis focuses on the creation of educational materials for teaching programming using computer graphics. The goal is to provide students with a set of practical tasks to enhance their programming skills. These tasks are graded and include not only the necessary theoretical explanations but also step-by-step instructions that enable independent work, making them ideal for practice. Automatic input data generators and validation programs offer students immediate feedback on their solutions. Additionally, students have access to a library of auxiliary functions. The tasks are designed to be universal and independent of any specific programming language. The practical application of these materials is demonstrated through examples in graphics, such as line rendering, polygon drawing, and anti-aliasing.

Keywords

programming education, computer graphics, educational materials, educational tasks, graduated tasks, line drawing, graphic algorithms, open-data tasks, Kotlin, anti-aliasing

Obsah

Úvod

V úvodu nejprve vysvětlím důvody pro volbu tématu práce, která se zaměřuje na vytvoření výukových materiálů pro programování, obsahujících úlohy z oblasti počítačové grafiky. Následně představím přístup, jenž tato práce využívá – rozšíření výuky o open-datové úlohy zaměřené na grafiku, které studentům poskytují atraktivní vizuální výsledky a propojují teoretické znalosti s praktickými dovednostmi.

Vzdělávání v oblasti programování často vyžaduje nejen teoretický základ, ale také praktické úlohy, které umožní studentům aplikovat nabyté znalosti a rozvíjet své dovednosti. Praktické procvičování by mělo být základní součástí výuky, protože studenty zapojuje do aktivního učení a pomáhá jim pochopit reálné souvislosti programátorských technik. Praktické úlohy poskytují široký kontext, ve kterém mohou studenti aplikovat naučené koncepty, a tím si osvojit nejen techniky psaní kódu, ale také logické myšlení a kreativní přístup k řešení problémů.

V současnosti existuje celá řada úloh zaměřených na procvičování programování. Mnohé z nich se však soustředí především na algoritmy a datové struktury. Tento přístup, ač důležitý, nepostihuje všechny možné oblasti, které by mohly být pro studenty přínosné a atraktivní. Jako příklad literatury, kde lze nalézt takto zaměřené úlohy, lze uvést „Töpfer: Algoritmy a programovací techniky“ [1] nebo „Mareš: Průvodce labyrintem algoritmů“ [2]. Tyto knihy poskytují cenné zdroje pro pochopení základních konceptů algoritmizace, ale jejich zaměření může být pro určité typy studentů příliš úzce profilované.

V této práci plánuji rozšířit spektrum úloh o jiné typy zadání, které by nejen pokrývaly technické aspekty programování, ale zároveň by studenty motivovaly svou atraktivitou. Konkrétně se chci zaměřit na úlohy z oblasti počítačové grafiky, které mají několik klíčových výhod:

- **Atraktivní výstupy**: Grafické úlohy často poskytují vizuální výsledky, například obrázky nebo animace, což může být pro studenty velmi motivující. Vidět konkrétní výsledek svého snažení na obrazovce má silný motivační efekt a podporuje zájem o další práci.
- **Existující podklady**: Oblast počítačové grafiky je dobře teoreticky zpracována a nabízí řadu publikací, z nichž lze při přípravě úloh vycházet. To usnadňuje vytváření zadání s jasně definovanými cíli a odpovídajícími výukovými materiály.
- **Jednoduchost základních algoritmů**: Základní grafické algoritmy, jako je například kreslení úsečky mezi dvěma body, jsou dostatečně jednoduché na to, aby je mohli implementovat i začínající studenti. Tyto úlohy tak představují ideální vstupní bod do programování, protože kombinují pochopení algoritmického myšlení s okamžitou zpětnou vazbou ve formě vizuálního výstupu.

Popsaný přístup by mohl podpořit rozmanitost výukových materiálů a nabídnout studentům nové možnosti, jak rozvíjet své schopnosti v programování. Počítačová grafika představuje atraktivní a praktickou oblast, kde se nejen dobře propojuje s výukou algoritmizace, ale také nabízí příležitost pro kreativní a zábavné zadání. Pro mladší žáky a studenty může být navíc velmi motivující, když mohou přímo vidět výsledky své práce. Úlohy zaměřené na grafiku jsou v tomto směru názornější a pro studenty lépe uchopitelné.

V našem případě se jedná o open-datové úlohy, což jsou programovací úlohy, které nevyžadují specifický programovací jazyk. Díky tomu jsou univerzální a přenosné mezi různými výukovými prostředími. Tento přístup umožňuje studentům zaměřit se na algoritmické řešení problému bez ohledu na konkrétní syntaxi jazyka. Open-datové úlohy navíc nabízejí možnost opakovaného generování náhodných vstupních dat, pro která lze spustit studentovo řešení a následně pomocí kontrolního programu (judge) vyhodnotit jeho správnost (judge typicky pro stejná vstupní data úlohu vyřeší a porovná svůj výstup s výstupem studenta). Studenti tak mohou snadno ověřit funkčnost svého kódu nezávisle na použití konkrétního jazyka. Open-datové úlohy podporují autonomii studentů, usnadňují jejich flexibilní využití ve výuce a lze je snadno integrovat do různých studijních programů.

Cíle

Cílem této práce je připravit komplexní sadu úloh, zaměřených na procvičení praktických dovedností programování. Tyto úlohy by měly studentům nabídnout možnost rozvíjet své schopnosti psaní kódu v reálných scénářích, které reflektují typické úkoly z praxe. Cíle práce jsou následující:

- **Vytvoření sady praktických úloh**: Každá úloha bude navržena tak, aby studenty krok za krokem provedla řešením reálných problémů. Úlohy se zaměří nejen na osvojení programátorských technik, ale i na rozvoj analytického a logického myšlení. Zvláštní důraz bude kladen na propojení teorie s praxí, aby si studenti mohli své znalosti vyzkoušet v různorodých situacích a získali představu o jejich využití v reálném světě.
- **Příprava výukových materiálů**: Součástí práce bude také tvorba podpůrných výukových textů, které studentům umožní úlohy řešit samostatně. Tyto materiály budou obsahovat:
  - Podrobné popisy zadání a jejich kontextu
  - Návrhy postupů řešení a vysvětlení klíčových konceptů, včetně potřebné teorie a algoritmů z počítačové grafiky
  - Ukázky správného řešení
- **Automatizované generování dat a vyhodnocení řešení**: Úlohy budou doplněny o možnost práce s náhodně generovanými vstupními daty, která si studenti mohou sami vygenerovat pomocí připraveného generátoru. Správnost svého řešení si následně ověří pomocí kontrolního programu (judge), jenž automaticky posoudí, zda výstup programu odpovídá očekávaným výsledkům.

Tento přístup podporuje nezávislost studentů při řešení úloh, posiluje jejich schopnost samostatně pracovat a zároveň poskytuje okamžitou a objektivní zpětnou vazbu.

- **Vytvoření podpůrných funkcí pro úlohy**: Součástí práce bude návrh a implementace podpůrných funkcí, které budou sloužit jako knihovna základních nástrojů pro práci s danými úlohami. Tyto funkce budou zaměřeny především na načítání a ukládání dat ve specifikovaném formátu a na jejich převod do podoby vizualizací, například obrázků. Podpůrné funkce umožní studentům snadno pracovat s datovými sadami definovanými v rámci úloh, aniž by se museli zabývat technickými detaily formátování či zpracování. Díky tomu mohou soustředit pozornost na samotnou logiku řešení problému, zatímco základní technická infrastruktura bude řešena těmito funkcemi.
- **Flexibilita zadání a gradované úlohy**: Úlohy budou koncipovány tak, aby umožňovaly diferenciaci podle úrovně dovedností studentů. Gradované zadání nabídne jednodušší základní variantu, kterou mohou zvládnout začátečníci, a postupně náročnější části, které budou výzvou pro pokročilé studenty. Tento přístup zajistí, že každý student dosáhne úspěchu odpovídajícího svým schopnostem a úrovni pokročilosti.
- **Univerzálnost úloh**: Úlohy budou koncipovány tak, aby nebyly svázány s konkrétním programovacím jazykem. Díky tomu je bude možné využít v různých výukových prostředích, ať už se jedná o Kotlin, Python, C++, Javu nebo jiné jazyky. Tato univerzálnost umožní snadné začlenění do jakéhokoliv vzdělávacího programu.

Hlavním výsledkem této práce by měla být ucelená sada výukových materiálů, která studentům nabídne nejen možnost rozvíjet jejich praktické dovednosti, ale také inspiraci pro další samostatnou práci. Díky zaměření na reálné scénáře a vizuálně atraktivní úkoly pomohou tyto úlohy zvýšit zájem studentů o programování a připraví je na úspěšné uplatnění v praxi.

Struktura

Struktura této práce je rozdělena do osmi hlavních kapitol. V úvodní části je představeno téma bakalářské práce, včetně motivace a důvodů pro výběr problematiky. Jsou zde popsány hlavní cíle práce a očekávaný přínos vytvořených vzdělávacích materiálů, které kombinují programování a počítačovou grafiku.

První kapitola poskytuje teoretický základ z oblasti počítačové grafiky. Podrobně vysvětluje rozdíly mezi rastrovou a vektorovou grafikou a popisuje různé barevné modely, jako jsou RGB, HSV, CMYK, LAB a Greyscale. Dále se věnuje specifikacím formátu BMP, včetně zjednodušené verze využívané v této práci, a klíčovému konceptu anti-aliasingu, který zlepšuje vizuální kvalitu grafických výstupů.

Druhá kapitola je zaměřena na open-datové úlohy, které tvoří jádro vzdělávacích materiálů. Vysvětluje princip těchto úloh, jejich rozdíly oproti closed-datovým úlohám a popisuje fungování kontrolního programu (judge). Dále se zabývá přínosy open-datových úloh ve výuce, zejména jejich flexibilitou a

nezávislostí na konkrétním programovacím jazyce.

Třetí kapitola představuje připravené vzdělávací materiály a jejich využití ve výuce. Jsou zde popsány konkrétní úlohy, jejich výhody a předpokládané znalosti studentů. Kapitola rovněž zahrnuje praktické rady pro přípravu programovacího prostředí a práci s nástroji, jako je Kotlin, který je využíván pro implementaci úloh.

Čtvrtá kapitola se zaměřuje na konkrétní úlohu „Vodorovná čára“. Obsahuje její zadání, popis generátoru vstupních dat a postup řešení krok za krokem. Kapitola je doplněna vzorovým řešením, které slouží jako příklad pro studenty a zároveň může posloužit jako návod, jak připravit vlastní open-datovou úlohu.

Pátá kapitola obsahuje zadání dalších úloh, které rozšiřují možnosti praktického procvičování. Mezi nimi jsou úlohy na vykreslení vodorovné a svislé čáry, čáry mezi dvěma body, obdélníku, vyplněného obdélníku a mnohoúhelníku, včetně úlohy na kreslení čáry s využitím anti-aliasingu. Každá úloha je doplněna podrobným popisem, návrhem postupu řešení a generátorem zadání.

Šestá kapitola se zaměřuje na vzorová řešení úloh. Obsahuje detailní popis řešení v Kotlinu, který studentům poskytuje inspiraci a pomáhá jim lépe pochopit danou problematiku.

Sedmá kapitola představuje knihovnu pomocných funkcí, která byla vytvořena jako součást této práce. Kapitola popisuje strukturu knihovny, jednotlivé funkce a jejich využití při řešení praktických úloh, což usnadňuje práci studentům i učitelům.

Závěrečná kapitola shrnuje dosažené výsledky a hodnotí přínos práce. Jsou zde také navržena možná rozšíření vzdělávacích materiálů a další směry vývoje, které mohou přispět k dalšímu zlepšení výuky programování s využitím grafiky.

Kapitola 1

Základy počítačové grafiky

1.1 Úvod

Moderní počítačová grafika se stala nedílnou součástí mnoha oblastí, od umělecké tvorby po technické obory. V digitálním světě je nezbytné rozumět základním principům a technologiím, stojícím za vizuální tvorbou a prezentací. Tento text se zaměřuje na klíčové pojmy, jako jsou rastrová a vektorová grafika, které představují dva rozdílné přístupy k zobrazení digitálního obrazu a nabízí přehled jejich výhod, nevýhod a specifického využití. Dále jsou uvedeny příklady různých barevných modelů, sloužících k přesné interpretaci barev v různých grafických systémech. Součástí textu je i téma antialiasingu, který je nezbytný pro hladké zobrazení hran a linií, což výrazně přispívá k estetice a zejména kvalitě výsledného obrazu.

Rozdíly mezi rastrovou a vektorovou grafikou popisuje [3] a tyto rozdíly si ukážeme v dalších částech.

1.2 Rastr vs. vektor

V dnešním světě grafiky se setkáváme s různými způsoby, jak zachytit obraz – od fotografií, které používáme na sociálních sítích, až po loga a ikony na webových stránkách. Dva hlavní způsoby, jak se obrazy vytvářejí a zobrazují, jsou vektorová a rastrová grafika. Každý z těchto přístupů má své specifické vlastnosti, určující, kde a jak se nejlépe využívají. Ať už jde o fotografie, které si chceme prohlížet v plném detailu, nebo o loga, která potřebujeme přizpůsobit různým velikostem, rozdíl mezi vektorem a rastrem je zásadní pro volbu správného formátu pro daný účel.

1.2.1 Vektorová grafika

Vektorová grafika je tvořena matematickými výrazy, které popisují tvary pomocí bodů, čar, křivek a polygonů. Každý tvar (objekt) je definován pomocí přesných geometrických údajů, jako jsou souřadnice a úhly. Díky tomu lze u vektorové grafiky měnit velikost tvarů nebo jejich rozmístění bez ztráty

kvality. Při zvětšování nebo zmenšování objektů se obraz nerozmazává a nejsou viditelné jednotlivé pixely, protože tvary se vždy přepočítávají podle matematických definic. To znamená, že jednotlivé tvary jsou vždy vykresleny přímo pro požadované rozlišení. Velikost vektorové grafiky je typicky menší než kvalitní obrázek v rastrové grafice. Další výhodou je také nenáročnost při úpravách. Jeden z nejvýznamnějších formátů využívající vektorovou grafiku je SVG (z anglického scalable vector graphics). Vektorová grafika je tedy ideální pro loga, ikony, text nebo jiné prvky, které je třeba často upravovat, nebo měnit jejich velikost. Se svými vlastnostmi je vektorová grafika nevhodná pro fotografie a z tohoto principu by bylo obtížně realizovatelné vytvoření vektorového obrázku na snímači fotoaparátu.

1.2.2 Rastrová grafika

Rastrová grafika (bitmapová grafika) se skládá z mřížky malých bodů nazývaných pixely. Každý pixel má definovanou barvu, a celkový obraz je tedy složen z barevných bodů, které tvoří vizuální dojem obrazu. Rastrová grafika je tak vhodná pro fotografie a obrázky s jemnými barevnými přechody a detaily. Nevýhodou je, že při zvětšení dochází k pixelizaci – obraz se stává rozmazaným, protože se zvětšují jednotlivé pixely a ty se poté stávají viditelnými. Rastrová grafika také podporuje složité barevné informace, což je užitečné pro realistické zobrazení. Nevýhodou oproti vektorové grafice je při zvyšování rozlišení také zvyšující se datová náročnost pro ukládání obrázku.

1.3 Barevné modely

V technickém světě se používá několik různých barevných modelů pro definici barev, přičemž každý model je vhodný pro jiný účel. Počítače nerozumí názvům jako „modrá“, „fialová“ nebo „lososově růžová“, a proto počítače používají číselné reprezentace barev, které jsou přesné a jednoznačné. Lidé často nemají jasnou představu o tom, jak vypadají konkrétní odstíny různých barev, pro které máme slovní označení. To je jeden z důvodů, proč se číselné modely staly nezbytností v digitálním prostředí.

Nejčastěji používané barevné modely jsou RGB (podsekce 1.3.1), HSV (podsekce 1.3.2) a CMYK (podsekce 1.3.3). Méně častý je model LAB (podsekce 1.3.4). Barevné modely, které si budeme níže představovat popisují „Color spaces for computer graphics“ [4], Photoshop LAB Color [5] a Greyscale and Colour Representation [6].

Každý barevný model má své specifické použití v závislosti na tom, zda se pracuje se světlem (například RGB pro displeje), pigmenty (CMYK pro tisk) nebo zda je prioritou lidské vnímání barev (HSV, LAB).

1.3.1 RGB

Jedním z nejpoužívanějších modelů je RGB model (Red – červená, Green – zelená, Blue – modrá), který funguje na principu aditivního míchání barev. To znamená, že jednotlivé složky se sčítají a výsledkem je světlo s vyšší či nižší intenzitou. Tento model je vhodný pro displeje, které produkují světlo (například monitory, televize). Každá složka (červená, zelená a modrá) má často 8bitovou hloubku, což znamená, že může nabývat hodnot mezi 0 a 255. Kombinací těchto tří složek lze vytvořit $(2^8)^3 = 16^6 = 16\,777\,216$ různých barevných odstínů, což je mnohem více než počet barev, které dokáže člověk pojmenovat.

Barevné hodnoty se mohou také vyjadřovat v šestnáctkové soustavě (hexadecimální), například barva s hodnotami RGB (68, 0, 255) bude reprezentována jako `#4400FF`. Tato reprezentace je často používána v oblasti webového designu a programování.

1.3.2 HSV

Další důležitý barevný model je HSV model (Hue, Saturation, Value), někdy označovaný jako HSB (Hue, Saturation, Brightness). Tento model je užitečný zejména pro uživatele, protože lépe odpovídá lidskému vnímání barev. Hue (odstín) určuje základní barvu (např. červená, zelená, modrá), Saturation (sytost) udává intenzitu barvy (čím vyšší sytost, tím méně šedé) a Value (jas) určuje, jak světlá nebo tmavá je barva. Tento model se používá při úpravách obrázků, protože umožňuje snadno měnit odstín, sytost a jas barev.

1.3.3 CMYK

Další model, CMYK model (Cyan, Magenta, Yellow, Key – černá), se používá v oblasti tisku. Oproti RGB, který pracuje s mícháním světla, tak CMYK využívá subtraktivní míchání barev, kde barvy vznikají odčítáním světla. Například při tisku na bílý papír se používají vrstvy inkoustu, které absorbují světlo různých vlnových délek. Když se smíchá modrozelená (cyan), purpurová (magenta) a žlutá (yellow), teoreticky by měly vytvořit černou. V praxi to ale nefunguje dokonale, proto se přidává černá složka (key), aby byla černá skutečně tmavá. To je zároveň vysvětlení, proč jsou v barevných tiskárnách typicky 4 cartridge. S větším množstvím cartridge se lze setkat u profesionálních tiskáren.

1.3.4 LAB

Zajímavou alternativou je také LAB model, který se zaměřuje na lidské vnímání barev. Tento model je speciálně navržen tak, aby byl více v souladu s tím, jak lidé vnímají barvy, a je užitečný v aplikacích, kde je potřeba přesně měřit barevné rozdíly.

Zajímavou alternativou je také barevný model LAB, který se používá v oblasti zpracování obrazu a grafického designu. Tento model, také označovaný jako CIELAB nebo jednoduše LAB, byl navržen Mezinárodní komisí pro osvětlení (CIE) a jeho účelem je reprezentovat barvy tak, aby co nejvíce odpovídaly lidskému vnímání. Na rozdíl od modelů RGB a CMYK, které závisí na zařízení, je LAB model založený na vnímání světla a jeho přenosu lidským okem, takže je považován za tzv. "device-independent"(nezávislý na zařízení). To znamená, že tento model umožňuje přesnou reprodukci barev mezi různými zařízeními, protože vychází z lidského vnímání barev a není závislý na konkrétním zobrazovacím nebo tiskovém zařízení. Nevýhodou je, že případný převod do jiných barevných modelů je výpočetně náročnější.

LAB model pro vyjádření barvy používá tři složky:

1. **L**: První složka (lightness – světlost) představuje světelnost neboli jas barvy, přičemž stupnice sahá od 0 (úplná černá) po 100 (čistě bílá).
2. **A**: Druhá složka určuje barvy na ose zelená–červená; záporné hodnoty značí zelené odstíny, zatímco kladné značí červené.
3. **B**: Třetí složka reprezentuje barvy na ose modrá–žlutá; záporné hodnoty odpovídají modrým tónům a kladné žlutým.

1.3.5 B/W

Model B/W, tedy černobílý (black-and-white), je barevný model s jedním kanálem, který zahrnuje pouze dvě barvy – černou a bílou. Tento model je často označován jako binární nebo monochromatický, protože funguje na principu přítomnosti či absence světla. Každý pixel v obrazu má pouze dvě možné hodnoty: 0 nebo 1, kde 0 reprezentuje černou barvu a 1 bílou barvu (nebo naopak, záleží na nastavení). Výhodou je zejména nízká paměťová náročnost a jednoduché použití. Tento barevný model díky své nízké paměťové náročnosti najde využití například při archivaci dokumentů, případně při tisku.

1.3.6 Greyscale

Greyscale (stupně šedi) model je barevný model, který využívá jeden 8bitový kanál pro reprezentaci odstínů šedé barvy, s rozsahem 256 úrovní. Hodnota 0 značí černou, 255 bílou a hodnoty mezi nimi představují různé odstíny šedé, což umožňuje plynulé přechody mezi světlými a tmavými tóny. Greyscale model se využívá v černobílé fotografii, lékařských snímcích (například rentgeny) a v obrazové analýze, kde je důležitý jas, ale ne barva.

Použití greyscale modelu přináší různé výhody, zejména:

- **Efektivní úložiště a nízká výpočetní náročnost**: Greyscale obrazy zabírají méně místa než barevné modely, ale stále poskytují detailní zobrazení. Menší množství dat má pak také výhodu

v rychlejším zpracování.

- **Přirozený kontrast**: Díky širokému rozsahu šedých tónů je greyscale ideální pro zobrazení světla a stínu.
- **Jednodušší použití**: Operace jako detekce hran jsou jednodušší, protože pracují jen s jedním kanálem.

1.4 Bitmapa

Nejjednodušší způsob, jakým lze uložit rastrový obrázek do souboru je použití formátu BMP (tzn. Windows Bitmap nebo zkrácením z device-independent bitmap také DIB). BMP ukládá data v binární podobě.

Výhodou použití BMP je jeho jednoduchost a zároveň rozšířenost napříč operačními systémy a různým grafickým softwarem. Formát BMP také není chráněn žádnou patentovou ochranou.

Nevýhoda, jejíž příčinu si vysvětlíme v další části, je výrazně větší velikost souboru než u jiných formátů. Tuto nevýhodu lze částečně kompenzovat použitím varianty s kompresí (např. použitím RLE – run length encoding), to však přináší větší komplexitu a náročnost pro použití BMP. Z této nevýhody tedy mj. plyne, že použití BMP není vhodné pro šíření internetem, neboť by zpomalovala načítání jednotlivých webových stránek.

Dále v textu budeme používat pro BMP také pojem bitmapa.

1.4.1 Specifikace formátu BMP

V této části při popisu formátu bitmapy budeme vycházet z knihy „Encyclopedia of graphics file formats“ ([7]). Formát bitmapy obsahuje několik základních částí

- **Bitmap file header** (hlavička souboru s bitmapou) – obsahuje obecné informace o souboru s obrázkem
- **DIB[1] header**: (DIB hlavička) – obsahuje detailní informace o souboru s obrázkem a definuje formát jednotlivých pixelů
- **Color table** (tabulka barev) – definuje barvy použité pro pole pixelů (viz podsekce 1.4.1), povinné pouze pokud barevná hloubka ≤ 8 bitů
- **Image data** (pole pixelů) – definuje hodnoty jednotlivých pixelů

Formát může také obsahovat několik dalších volitelných částí (např. definice pixelů, barev anebo zarovnání struktury pro optimalizaci binárního souboru). Těmito částmi se však pro zjednodušení nezabýváme. Nyní si podrobněji rozebereme povinné části.

[1] device-independent bitmap

Bitmap file header

Hlavička souboru je podobná jako u ostatních formátů. Specifikuje se zde, o jaký formát se jedná a je zde také uvedena velikost souboru. Programy sloužící pro otevření BMP souboru tak mohou identifikovat, že se jedná právě o BMP. Na základě toho pak během otevírání souboru daný software ví, jak zpracovat data následující po hlavičce souboru.

DIB header

DIB hlavička obsahuje detailní vlastnosti pro daný obrázek. Jedná se například o šířku a výšku obrázku, velikost jednotlivých pixelů v bitech, počet použitých barev anebo počet bitů pro pixel.

Color table

Tabulka barev je v některých případech volitelná, což se odvíjí na základě bitové velikosti daného souboru. V případě velikosti ≤ 8 bitů se zde definují jednotlivé barvy a na ty se poté odkazuje pomocí indexu. V případě velikosti ≥ 16 bitů je tabulka barev volitelná a může obsahovat informace o tom, kolik bitů používají jednotlivé RGB složky.

V případě 16 bitové velikosti tedy například: červená 5 bitů, zelená 6 bitů a modrá 5 bitů, součet bitů je 16 což odpovídá bitové velikosti a zároveň je zde větší množství zelených odstínů, které lidské oko vnímá nejcitlivěji.

Image data

Image data v BMP tak poskytují podrobný návod, jak vykreslit každý pixel obrázku. Část image data obsahuje informace o barvách každého pixelu obrázku. Pixely jsou uspořádány do řádků a sloupců a každý pixel je v BMP reprezentován hodnotou, která odpovídá jeho barvě. Většina BMP souborů má barvy pixelů zakódovány pomocí RGB modelu. V takovém případě má každá barva určitou hodnotu v rozsahu od 0 do 255 (v případě jiných barevných modelů může být rozsah i jiný).

Hodnoty každého pixelu jsou uloženy ve specifickém pořadí (v případě RGB modelu obvykle ve formátu „BGR“), kde první je hodnota modré barvy, pak zelené a nakonec červené. BMP ukládá pixely odspodu nahoru, takže první řádek dat image data představuje spodní část obrázku, zatímco poslední řádek reprezentuje horní část.

Image data musí být zarovnána na násobky 4 bajtů (4-byte alignment). To znamená, že pokud počet bajtů v řádku není dělitelný čtyřmi, přidají se tzv. padding bajty (jako výplň), aby se dosáhlo správného zarovnání. Použití paddingu je běžné při binárním ukládání dat, protože zajišťuje, aby byla data uložena na adresách, které vyhovují požadavkům procesoru na zarovnání. U BMP formátu to

znamená, že každý pixel je zarovnán na adrese dělitelné čtyřmi (bajty), což umožňuje 32bitovým procesorům rychlejší zpracování dat, protože pracují nejefektivněji, pokud přistupují k paměti právě na adresách dělitelných čtyřmi. Tento princip je univerzální – u 8bitových, 16bitových a dalších procesorů se využívá zarovnání na mocniny dvou, aby bylo čtení a zápis dat optimální. Zarovnání na násobky čtyř tak usnadňuje práci s daty, protože díky němu každý nový řádek vždy začíná na adrese dělitelné čtyřmi. Bity paddingu však neobsahují žádná užitečná data o obrázku a nemají vliv na jeho vzhled; slouží výhradně k zarovnání a zabírají místo v paměti. Přínos efektivnějšího přístupu k paměti však převažuje nad drobným zvýšením velikosti souboru.

1.4.2 Zjednodušená bitmapa pro naše potřeby

Pro naše potřeby bude jednodušší pokud si definujeme vlastní formát pro bitmapu v textové podobě, budeme mít pouze jednu hlavičku a samotná data (podobně jako image data – viz podsekce 1.4.1).

Formát bitmapy využívané pro účely naší práce:

1. řádek obsahuje hlavičku ve formátu [typ],[barevný model], tedy například **BMP,GREY**. Další hodnoty pro *barevný model* mohou být například RGB, HSV, BW atd.

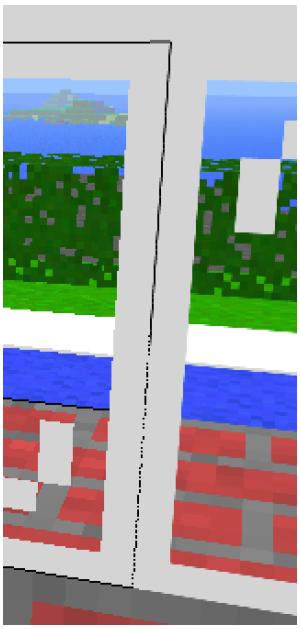
2. řádek obsahuje rozměry obrázku s bitmapou ve formátu [šířka] [výška], tedy například **10 5**. Šířku budeme v práci nazývat w (odvozeno od anglického **w**idth) a výšku h (odvozeno od anglického **h**eight)

Na dalších řádcích následuje h řádků, kde jsou jednotlivé pixely oddělené mezerou. Počet pixel na jednom řádku je w. Tedy například:

```
0 0 255 255 255 0 0 0 0 0

0 255 255 0 255 0 0 0 0 0

0 255 0 0 255 255 0 0 0 0

0 255 0 0 0 255 255 0 0 0

0 0 255 0 0 0 255 255 0 0
```

1.5 Anti-aliasing

Představte si, že se díváte na obrázek nebo hrajete hru, kde jsou hrany objektů neostré a vypadá to, jako by byly poskládané z drobných schůdků nebo kostek. Tento efekt je způsoben tím, jak obrazovka zobrazuje jednotlivé body neboli pixely – malé čtverečky, které společně tvoří celý obraz. Když se



Obrázek 1.1: Porovnání vykreslené scény bez (levá strana) a s (pravá strana) aplikovaným supersampling anti-aliasingem[8]

hrana objektu nekryje přesně s řadami těchto pixelů, objeví se „zubaté“ nebo kostičkované hrany, které ruší celkový dojem. Říká se tomu aliasing a právě k odstranění tohoto kostičkovaného efektu slouží metoda zvaná anti-aliasing

Anti-aliasing využívá princip, že místo toho, abychom kreslili pixely pouze v extrémních barvách, jako je černá a bílá, přidáváme k hranám také odstíny šedé. Tento přechod mezi barvami pomáhá vytvořit jemné a plynulé rozhraní, čímž eliminuje viditelné zubaté okraje, které by jinak vznikly. Například v algoritmu Xiaolina Wua pro vykreslování čar jsou pixely blíže k hraně vykresleny s nižší intenzitou barvy, čímž vzniká plynulý přechod mezi barvami. Výsledkem je, že hrany a objekty vypadají hladce a přirozeně, což výrazně zlepšuje vizuální kvalitu vykreslených scén. Tato technika je zvláště důležitá pro grafiku, kde detaily a jemnost mají velký vliv na celkový dojem.

Příklady použití anti-aliasingu

- **Text a textové fonty**: Anti-aliasing se používá při vykreslování jednotlivých znaků textu. Jednotlivá písmena jsou poté lépe čitelná a nepůsobí rušivě ostrými a hrubými okraji (viz Obrázek 1.2 a Obrázek 1.3).
- **Počítačové hry**: Anti-aliasing při pohybu postav a objektů v herním světě zajišťuje, že hrany nebudou působit rozpixelovaně nebo zubatě (viz Obrázek 1.1).
- **Grafické návrhy**: Vyhlazení hran je důležité při tvorbě ilustrací, návrhů a digitálních obrazů – díky němu působí výsledný obraz hladce a profesionálně.

Obrázek 1.2: Porovnání písmen „G W a“ s detailním přiblížením, nahoře s anti-aliasing, dole bez anti-aliasing (vlastní tvorba)

Obrázek 1.3: Porovnání zvětšených písmen „G W a“, nahoře s anti-aliasing, dole bez anti-aliasing (vlastní tvorba)

V této práci se budeme zabývat anti-aliasingem v kontextu algoritmu Xiaolina Wua (v angličtině známý jako Xiaolin Wu's algorithm), který slouží k vykreslování jemně vyhlazených čar v počítačové grafice. Anti-aliasing v tomto případě zajišťuje, že čára nebude působit kostrbatě díky postupnému přechodu mezi pixely různých jasů. Wuův algoritmus tedy přidává jemné odstíny okolo čáry, což vytváří plynulejší a přirozenější vzhled a zvyšuje kvalitu vizuální prezentace.

Anti-aliasing je klíčová technika pro dosažení kvalitního vykreslení grafiky. Jako zajímavý příklad lze zmínit Apple, který začal používat anti-aliasing u ikon už v roce 1983 na svém počítači Apple Lisa ([9]), jednom z prvních osobních počítačů s grafickým rozhraním. Tehdejší monitory měly nízké rozlišení, a proto Apple hledal způsoby, jak ikony zjemnit, aby působily hladce a čitelněji. Přidáním anti-aliasingu vypadaly hrany hladší a celé rozhraní působilo vizuálně kvalitněji. Počítač Lisa tak využíval anti-aliasing nejen pro estetičtější vzhled, ale i pro vytvoření dojmu vyššího rozlišení – i když fyzické rozlišení obrazovky zůstalo stejné. Díky anti-aliasingu získaly ikony hladší přechody, takže na obrazovce působily jemněji, a uživatelé mohli pracovat s přehlednějším rozhraním. Tento přístup později zdokonalil Macintosh, který v roce 1984 přinesl ještě více vylepšení ve vykreslování ikon a uživatelského rozhraní

Kapitola 2

Open-datové úlohy a jejich fungování

V této kapitole se zaměříme na praktické programátorské úlohy a zejména na úlohy open-datové. Nejprve si vysvětlíme, co jsou praktické programovací úlohy a jejich rozdělení (sekce 2.1). Pak si podrobněji popíšeme jejich fungování (sekce 2.3). Dále popíšeme motivaci pro používání praktických programátorských úloh ve výuce (sekce 2.4), zejména open-datových úloh, jejichž hlavní výhodou je, že jejich řešení není závislé na konkrétní technologii, takže studenti mohou využít programovací jazyk podle jejich libosti (sekce 2.5). Na závěr této kapitoly zmíníme, kde se open-datové úlohy často používají (sekce 2.6). Principem open-datových úloh se zabývá mimo jiné web KSP [10] ze kterého čerpáme některé informace v této kapitole.

2.1 Co je praktická programovací úloha?

V programování se můžeme často setkat s různými praktickými programovacími úlohami. Tyto úlohy často odrážejí reálné problémy, které vyžadují práci s velkým objemem dat nebo komplexními algoritmy. Praktické úlohy se typicky zaměřují na zpracování vstupních dat (například ve formě textových souborů nebo jiných datových formátů), tato data jsou následně zpracována programem řešící danou úlohu a na základě výstupu z programu se ověří, zda program úlohu řeší správně nebo ne. Praktické programovací úlohy se dají rozdělit na dva hlavní druhy, říkejme jim closed-datové úlohy a open-datové úlohy.

Closed-datové úlohy

Closed-datové úlohy předpokládají, že student odevzdá svůj program skrze nějaký systém, který sám vygeneruje data, spustí s těmito daty program a ověří výstup programu. Celý tento proces se odehrává uvnitř systému a student má možnost pouze odevzdat program a zobrazit si, zda program zpracovává data správně nebo ne. Tento přístup je nevýhodný v tom, že systém musí umět spouštět programy v různých jazycích, množina těchto jazyků může být malá a studenti jsou při výběru programovacího jazyka omezeni. Údržba takového systému může být značně náročnější. Studenti navíc nemají přístup ke vstupním datům, takže se jim obtížněji hledá chyba a nemohou s danými daty program debugovat, tato vlastnost ale může být i výhodou a to tím, že studenti musí více přemýšlet nad zadáním, nad tím,

jaká různá data mohou dostat a jaké okrajové scénáře ohlídat. Tento systém ale může kontrolovat i další vlastnosti programu – například zda je dostatečně rychlý a efektivní.

Open-datové úlohy

Naopak open-datové poskytují studentům větší svobodu. Systém v open-datových úlohách se zaměřuje jen na vygenerování vstupních dat a kontrolu výstupu. Vstupní data si student vygeneruje například lokálně staženým programem nebo si je odněkud stáhne. Poté spustí program řešící danou úlohu, studentův program vygeneruje výstupní data a výstupní data jsou opět poskytnuty systému, který zkontroluje, zda jsou správná. Systém může být lokální kontrolní program nebo nějaký vzdálený server, který poskytuje pro kontrolu úlohy rozhraní (grafické, REST API atp.). Tento přístup na rozdíl od closed-datových úloh umožňuje naprostou nezávislost na tom, jaký programovací jazyk student použije. Vzhledem k tomu, že ale studenti mají přímý přístup ke vstupním datům, tak je nutné mít i generátor vstupních dat, který bude generovat pokaždé jiná vstupní data. Oproti tomu v closed-datových úlohách teoreticky stačí mít jen jednu variantu vstupních dat, protože je student nevidí a nemůže tak vyřešit okrajové případy způsobem, který by nebyl obecný pro všechna možná vstupní data.

2.2 Princip praktických programovacích úloh

V této kapitole si postupně představíme klíčové části praktických programovacích úloh. Řešení praktické programovací úlohy se typicky skládá ze tří částí:

- vstupní data (podsekce 2.2.1) a jejich načtení
- zpracování dat (podsekce 2.2.2) – program řešící úlohu
- vypsání výstupu (podsekce 2.2.3)

2.2.1 Vstupní data

Praktické programovací úlohy využívají různé formáty vstupních dat, jako jsou textové soubory, JSON, CSV nebo XML. Tyto soubory obsahují strukturovaná data, která program musí umět načíst a podle zadání zpracovat. Vstupní data mohou být různorodá – od jednoduchých seznamů až po složitější struktury s různě propojenými informacemi, například jednoduchými textovými databázemi.

V zadání každé praktické programovací úlohy je formát vstupních dat vždy přesně popsaný. Úkolem programátora je napsat kód, který tento formát umí načíst a zpracovat podle pokynů v zadání. Kód programu tedy nemusí „hádat“, co jednotlivé hodnoty znamenají. Každý údaj je totiž jednoznačně definován a není nutné odhadovat, co jednotlivé hodnoty znamenají. Programátor se ale musí postarat o to, aby program správně interpretoval strukturu těchto dat, například jak jsou uspořádána a která informace s čím souvisí.

V praxi to znamená, že pokud zadání obsahuje data ve formátu JSON s údaji o jednotlivých osobách, program musí vědět, že například pod klíčem „jméno“ najde jméno osoby, a že data o adresách jsou uložena v jiné části. Důležité tedy je napsat kód tak, aby dokázal pracovat s daty přesně podle popsané struktury a zvládl načíst i složitější formáty, kde jsou data třeba vnořená nebo navzájem propojená.

Vstupní data mohou obsahovat různé druhy informací, například:

1. **Jednoduché seznamy** – jako seznam čísel, jmen nebo názvů, které není potřeba vzájemně spojovat.
2. **Tabulková data** – podobná tabulkám v aplikacích jako Excel (například CSV formát), kde data obsahují sloupce a řádky, například s údaji jako „jméno, věk, město“.
3. **Hierarchická data** – často uložená ve formátech jako JSON nebo XML, kde jsou data organizována do „úrovní“; například u osoby mohou být kromě jména a věku ještě další podrobnosti jako adresa, a adresa může mít vlastní dílčí části (ulici, město atd.).
4. **Komplexní data** – například grafy nebo sítě, kde mohou být data vzájemně propojená (například vztahy mezi uzly grafu, jako jsou trasy mezi městy nebo vztahy v sociálních sítích, tok v sítích).

Kód u praktických programovacích úloh musí být dostatečně flexibilní, aby uměl data správně načíst a zpracovat i bez pevné struktury předem definované v programu. Musí tedy podle zadání určit, co která data znamenají a jak s nimi pracovat. Díky tomu se můžeme učit, jak psát programy, které zpracují různé typy dat, aniž by bylo nutné ručně data přepisovat nebo měnit kód podle konkrétního formátu. Příklad této vlastnosti uvidíme v práci například během zpracování našeho formátu bitmapy – na základě hlavičky, která určuje jaký barevný model budeme používat při jednotlivých spuštěních programu na různých vstupních datech. V případě BW modelu pracuje program pouze s jednou hodnotou (bílá nebo černá). Oproti tomu při použití RGB modelu program pracuje s třemi čísly (jednotlivé barevné složky R, G, B).

2.2.2 Zpracování dat

Řešitel vytváří algoritmus, který načte vstupní data, analyzuje jejich strukturu a provede na nich požadované operace. Tento krok může zahrnovat například filtrování, třídění, výpočty, hledání specifických vzorců či optimalizací. Důležité je, aby algoritmus byl navržen efektivně, zejména pokud vstupy obsahují velké množství dat, která by mohla zpomalit výkon programu.

2.2.3 Výstup

Na základě zpracování vstupních dat musí program vygenerovat výstup ve specifikovaném formátu, který odpovídá zadání. Tento výstup může být číslo, řetězec, soubor nebo jiná struktura. Hlavním cílem je, aby byl výstup korektní a odpovídal požadavkům úlohy, což obvykle vyžaduje přesné pochopení a

implementaci pravidel, definovaných v zadání úlohy.

Praktické programovací úlohy jsou oblíbené nejen kvůli jejich praktickému zaměření, ale také proto, že podporují práci s reálnými datovými formáty a procvičují schopnost efektivně analyzovat a pracovat s velkými objemy informací. Jsou ideálním prostředkem pro rozvoj dovedností, které jsou v programování často vyžadovány v praxi.

2.3 Fungování a princip kontrolního judge u open-datových úloh

V této části se zaměříme na kontrolu správnosti řešení praktické open-datové úlohy. Bez ohledu na to, zda kontrolu budeme provádět lokálně na svém PC, nebo kontrola bude prováděna na vzdáleném serveru, tak budeme potřebovat program, který zkontroluje výstupní data. Takovému programu v naší práci budeme říkat „judge“.

Judge lze rozdělit na dva základní typy:

1. **Porovnávací judge**: Tento typ judge je častější a spočívá v tom, že danou úlohu sám vyřeší a následně porovná zda se toto řešení shoduje s řešením, které poskytl řešitel úlohy.
2. **Kontrolní judge**: Tento typ judge se typicky používá u úloh, kde je více správných řešení. Taková úloha by třeba požadovala, aby výstupem bylo náhodné sudé číslo – v takovém případě existuje nekonečno správných řešení. Kdyby náhodné číslo vygeneroval porovnávací judge, tak se s velmi vysokou pravděpodobností nebude shodovat s náhodným číslem od studenta. Proto tento typ judge provede kontrolu, zda je číslo dělitelné 2. Pokud ano, řešení je správné.

V průběhu naší práce se budeme setkávat s druhým typem judge – kontrolním.

2.4 Motivace pro použití praktických programovacích úloh

Motivace pro použití praktických programovacích úloh v programování spočívá v několika klíčových aspektech, které z nich dělají efektivní nástroj jak pro výuku, tak i pro praktické uplatnění v reálném světě. Tyto úlohy odrážejí výzvy, se kterými se programátoři setkávají v praxi. Současně také poskytují prostor pro kreativitu, řešení problémů a rozvoj kritického myšlení.

2.4.1 Práce s reálnými daty

Praktické programovací úlohy často simulují scénáře, které zahrnují práci s daty z reálného světa, jako jsou textové soubory, datové sady v různých formátech, statistiky nebo logy. Tím se studenti a další řešitelé úloh připravují na situace, se kterými se mohou setkat při vývoji aplikací, datové analýze

nebo automatizaci procesů. Řešení takových úloh vyžaduje schopnost porozumět datům, správně je interpretovat a navrhnout algoritmy pro jejich efektivní zpracování.

2.4.2 Rozvoj algoritmického myšlení

Dalším motivačním faktorem je potřeba vytvářet efektivní algoritmy pro analýzu a zpracování dat. Praktické programovací úlohy často kladou důraz na efektivitu řešení, protože vstupy mohou být velké a náročné na zpracování. Programátoři jsou tak nuceni zvažovat časovou a prostorovou složitost svých algoritmů a optimalizovat své kódy, což rozvíjí jejich algoritmické myšlení, schopnost efektivně analyzovat a řešit složité problémy, a zároveň podporuje dovednost rozložit problém na menší části a nalézt pro něj efektivní řešení.

2.4.3 Kreativní přístup k řešení problémů

I když mají praktické programovací úlohy jasně definované vstupy a výstupy, způsob, jakým je možné dosáhnout správného výsledku, bývá flexibilní. To poskytuje prostor pro kreativitu a volbu různých přístupů k řešení. Uživatelé často mohou využívat různé programovací techniky, datové struktury nebo optimalizace podle toho, co považují za nejvhodnější. Tento aspekt je nejen motivující, ale také podporuje rozvoj dovednosti přistupovat k problémům z různých úhlů pohledu.

2.4.4 Reálná zpětná vazba

Jednou z výhod praktických programovacích úloh je jejich schopnost poskytovat okamžitou zpětnou vazbu. Po zpracování vstupních dat je možné okamžitě ověřit, zda program generuje správný výstup, což mnohdy motivuje účastníky k opakovanému pokusu o optimalizaci a zlepšení svých řešení. Tento cyklus pokusů a zpětné vazby je klíčovým prvkem, který podporuje kontinuální učení.

2.4.5 Příprava na reálné programátorské výzvy

V reálném světě se programátoři často setkávají s problémem práce s rozsáhlými nebo nestrukturovanými daty. Open-datové úlohy připravují řešitele na takové scénáře tím, že jim poskytují komplexní datové vstupy, které vyžadují analýzu, transformaci a extrakci relevantních informací. Tento trénink pomáhá rozvíjet dovednosti, které jsou nezbytné při práci s databázemi, datovými analýzami nebo při vytváření softwaru pro práci s velkými objemy dat.

2.4.6 Zvýšení zájmu a motivace studentů

Ve vzdělávání mohou praktické programovací úlohy hrát zásadní roli při zvyšování zájmu studentů o programování. Díky tomu, že úlohy představují praktické problémy, mohou studenti lépe pochopit, jak znalosti programování uplatnit v reálných situacích. Řešení takových úloh bývá často zábavné a přináší uspokojení z úspěšného dokončení výzvy, což může motivovat k dalšímu rozvoji.

Díky těmto důvodům jsou praktické programovací úlohy široce používány jak v soutěžích, tak i ve vzdělávacích programech a v rámci praxe, kde podporují hlubší porozumění programovacím technikám a přístupu k datům.

2.5 Výhody open-datových úloh

Open-datové úlohy představují moderní a flexibilní nástroj, který umožňuje studentům pracovat s reálnými daty, což zvyšuje praktickou hodnotu jejich úsilí. Hlavním přínosem tohoto přístupu je, že studenti mohou řešit problémy, které nejsou pouze teoretické, ale odrážejí skutečné výzvy z různých oblastí. To vede k hlubšímu pochopení, jak lze algoritmy a programovací postupy využívat k řešení aktuálních problémů.

Jednou z hlavních výhod open-datových úloh je jejich nezávislost na konkrétních technologiích. Úlohy mohou být navrženy tak, aby byly řešitelné v jakémkoli programovacím jazyce nebo nástroji, což poskytuje studentům možnost volby, jakým způsobem k řešení přistoupí. Různé jednodušší úlohy dokonce mohou být vyřešeny i v prostředích, jako je Excel, nebo dokonce na papíře, pokud jsou zadána vhodná vstupní data. Tato rozmanitost open-datových úloh může být obzvláště cenná například ve výuce, kde se úrovně dovedností jednotlivých studentů mohou výrazně lišit.

Využití open-datových úloh přináší do výuky několik dalších výhod:

1. **Přístupnost**: Studenti nejsou limitováni konkrétní technologií nebo programovacím jazykem. Mohou se rozhodnout pracovat v prostředí, které jim nejlépe vyhovuje, což podporuje kreativitu a individuální přístup k řešení problémů. Tato přístupnost je důležitá nejen v rámci samotné výuky, ale i pro reálné využití těchto dovedností mimo školu.
2. **Generované vstupy**: Vstupy pro tyto úlohy mohou být snadno generovány, což umožňuje vytvářet velké množství různých zadání, aniž by se opakovala stejná data. To znamená, že každý student může mít své vlastní zadání, což zabraňuje kopírování řešení a podporuje individuální učení. Generování různých vstupů rovněž umožňuje učitelům přizpůsobit úlohy konkrétním výukovým cílům a úrovním studentů.
3. **Široké využití**: Open-datové úlohy nacházejí uplatnění i napříč různými obory. Tento typ úloh je snadno přizpůsobitelný různým oblastem, od analýzy dat přes simulace až po vývoj softwaru. Ve výuce to umožňuje učitelům vytvořit různorodé úkoly, které mohou studenty zapojit do řešení

problémů s přesahem do reálného světa.

4. **Neomezený přístup**: Díky tomu, že open-datové úlohy nejsou svázány s žádným konkrétním prostředím, mohou studenti pracovat kdykoli a kdekoli. To podporuje samostatné učení a umožňuje studentům vracet se k úlohám i mimo školní hodiny. Taková nezávislost nejen podporuje vlastní tempo učení, ale i dlouhodobější zájem o danou problematiku.
5. **Reálný kontext**: Práce s reálnými daty, která jsou volně dostupná, studentům ukazuje skutečný význam jejich dovedností. Mohou například analyzovat data o klimatu, finančních trzích nebo demografických změnách, což převádí učení do kontextu reálného světa a zvyšuje motivaci. Tento aspekt lze využít i pro propojení s jinými předměty, jako jsou přírodní vědy či společenské vědy.

Open-datové úlohy tak přinášejí do výuky programování nejen technickou flexibilitu, ale i možnost pracovat na úlohách, které mají reálný dopad. To studenty motivuje k hlubšímu učení, protože vidí přímý vztah mezi tím, co se učí, a tím, jak mohou své znalosti využít ve světě mimo učebnu.

2.6 Rozšíření open-datových úloh

S open-datovými úlohami se můžeme velice často setkat například v programátorských soutěžích, jako je Advent of Code[11], kde jsou denně během adventu zadávány nové úlohy k řešení. Z českých soutěží využívají open-datové úlohy například Kasiopea[12] nebo KSP[13].

Svým způsobem za open-datové úlohy můžeme považovat i vytvoření aplikací, které čerpají z otevřených dat a zpracovávají je do uživatelsky přívětivějšího rozhraní nebo různá otevřená data kombinují a na základě toho poskytují různé výstupy. S otevřenými daty se v Česku můžeme setkat například u datových sad[14].

Kapitola 3

Připravené materiály a jejich využití ve výuce

Tato kapitola se zaměřuje na využití počítačové grafiky jako nástroje pro výuku programování. Představuje sadu gradovaných praktických úloh, které studenty postupně vedou od základních technik k pokročilejším algoritmům. Úlohy jsou navrženy tak, aby podporovaly samostatnou práci studentů, kombinovaly teoretické základy s praktickými ukázkami a poskytovaly okamžitou vizuální zpětnou vazbu.

Podrobně zde popisujeme připravené materiály pro výuku programování s využitím počítačové grafiky (viz sekce 3.1). Věnujeme se výhodám praktických gradovaných úloh (viz sekce 3.2), využití gamifikace ve výuce (viz podsekce 3.2.1) a přípravě potřebného programovacího prostředí (viz sekce 3.6 a sekce 3.4). Kapitola obsahuje také pokyny pro učitele (viz sekce 3.5), jak materiály začlenit do výuky a přizpůsobit je různým úrovním znalostí studentů, v závěru pak návod pro studenty (viz sekce 3.7), jak úlohy efektivně řešit.

3.1 Připravené materiály pro výuku programování s využitím počítačové grafiky

Pro tuto práci jsem připravila sadu gradovaných praktických úloh, které lze využít při výuce programování, zejména pro samostatné procvičování. Úlohy jsou navrženy tak, aby studenty postupně vedly od základních technik ke složitějším problémům v oblasti rastrové grafiky. Úlohy se zaměřují na vykreslování základních geometrických tvarů, jako jsou čáry, obdélníky a mnohoúhelníky, na černobílé bitmapě. Každá úloha je pečlivě strukturovaná, obsahuje krokový postup řešení a zadání se specifikací vstupních parametrů, i podrobný návod k jejich zpracování. Hlavním cílem je podpořit samostatnou práci studentů, proto jsou úlohy doplněny základními výukovými materiály. Tyto materiály obsahují přehled potřebné teorie, příklady použití a návod, jak k řešení přistupovat. Díky tomu mohou studenti úlohy řešit samostatně bez nutnosti přímé asistence, což posiluje jejich schopnost pracovat samostatně

a nacházet vlastní řešení. Sada úloh je vhodná jak pro školní hodiny, tak pro domácí procvičování. Jejím cílem je nejen rozvíjet praktické dovednosti v programování, ale také studentům ukázat, jak lze teoretické znalosti aplikovat na konkrétní problémy. Díky postupnému zvyšování obtížnosti jsou úlohy vhodné pro různé úrovně pokročilosti.

3.1.1 Seznam úloh

1. Vodorovná čára(sekce 4.1)
2. Vodorovná a svislá čára(sekce 5.1)
3. Čára v libovolném směru mezi 2 body(sekce 5.2)
4. Obdélník(sekce 5.3)
5. Vyplněný obdélník(sekce 5.4)
6. Mnohoúhelník(sekce 5.5)
7. Kreslení čáry s anti-aliasingem(sekce 5.6)

3.2 Výhody praktických gradovaných úloh

Počítačová grafika studentům poskytuje prostředí pro procvičování algoritmizace a základních programovacích principů. Popsaný přístup kombinuje vizuální a praktické aspekty, což studenty směřuje k řešení problémů a aplikaci naučených technik na reálné projekty.

Abstraktní problémy mohou studenty snadno odradit, pokud nevidí okamžité výsledky. Počítačová grafika však nabízí smysluplné projekty, které zvyšují jejich angažovanost a umožňují praktické zkušenosti s algoritmy. Vizuální výstupy kódu poskytují zpětnou vazbu, která posiluje porozumění a zvyšuje sebedůvěru. Kombinace teoretických a praktických aspektů výuky pak motivuje studenty k aplikování znalostí v reálných situacích. Experimentování s grafikou a studium algoritmů urychluje rozvoj jejich schopností a podporuje aktivní zapojení do výuky.

V rámci této kapitoly se podrobně zaměříme na soubor úloh, které byly pečlivě navrženy tak, aby studentům poskytly příležitost procvičovat a zdokonalovat různé algoritmické přístupy. Každá úloha je koncipována tak, aby reflektovala běžně používané techniky a principy, které jsou klíčové pro pochopení a aplikaci algoritmického myšlení.

Jedním z hlavních přínosů těchto úloh je jejich postupná gradace, což znamená, že obtížnost úloh se postupně zvyšuje. Studenti začínají jednoduššími úkoly, které slouží jako úvod do problematiky a umožňují jim seznámit se se základními koncepty a metodami. Tyto úvodní úlohy jsou navrženy tak, aby byly přístupné a motivující, což pomáhá studentům vybudovat si sebevědomí a základní dovednosti.

Jakmile studenti zvládnou základní úkoly, postupně přecházejí k složitějším výzvám, které vyžadují

hlubší analytické myšlení a schopnost aplikovat naučené techniky na komplexnější scénáře. Tímto způsobem se studenti nejen seznamují s různými algoritmickými přístupy, ale také si rozvíjejí schopnosti kritického myšlení a problémového řešení, které jsou nezbytné pro úspěšné zvládnutí pokročilejších témat v oblasti informatiky a programování.

3.2.1 Gamifikace jako motivátor ve výuce

Gamifikace ve výuce přináší nejen zábavný, ale také efektivní způsob, jak udržet studenty zapojené a motivované. Studie Thomase Malone ([15]) definuje pět základních charakteristik optimálního vzdělávacího prostředí s herními prvky:

- Volný pohyb prostředím podle zkušeností a přizpůsobení zkušenostem.
- Získání okamžité zpětné vazby o výsledcích.
- Postup vlastním tempem podle individuálních potřeb.
- Neexistence omezení prostředím při objevování.
- Nabádání k objevování souvislostí.

Tyto principy se ve výuce programování uplatňují především v gradovaných úlohách, které umožňují studentům určovat si vlastní tempo a vybírat si úlohy odpovídající jejich aktuálním dovednostem.

Gradace obtížnosti úloh a různorodost možných cest výuky přizpůsobují učební proces individuálním schopnostem studentů. Zatímco pokročilejší mohou soutěžit o to, kdo vyřeší více úloh, či usilovat o rychlé dosažení vyšších úrovní, začátečníci se mohou soustředit na získání pevných základů. Tím je zajištěno, že každý student má dříve nebo později šanci dosáhnout úspěchu podle svých možností. Díky použití open-datových úloh studenti mají okamžitou zpětnou vazbu, což opět motivuje studenty k dalšímu zlepšování a pomáhá jim lépe vnímat svůj pokrok.

3.3 Předpokládané znalosti studentů

Tato práce vychází z předpokladu, že studenti mají základní znalosti programování, což zahrnuje alespoň základy jednoho programovacího jazyka. Není důležité, o jaký konkrétní jazyk se jedná, protože úlohy v této práci jsou navrženy tak, aby je studenti mohli řešit v libovolném jazyce, který si zvolí nebo který jim doporučí jejich učitel. Tato flexibilita umožňuje každému přistoupit k úlohám s jazykem, který je mu blízký, nebo ve kterém má již určité zkušenosti.

Co studenti potřebují umět před začátkem práce s úlohami:

1. **Deklarace proměnných a datové typy** – Studenti by měli rozumět tomu, jak se v jejich jazyce vytvářejí proměnné a jak s nimi mohou pracovat. To zahrnuje i znalost různých datových typů,

jako jsou čísla, textové řetězce, logické hodnoty (`true`/`false`) a další základní typy, které jim umožní pracovat s různými typy informací.

2. **Řízení běhu programu** – Dalším předpokladem je schopnost pracovat s podmínkami a cykly:
   - **Podmínky** – Studenti by měli umět použít podmínkové příkazy, které rozhodují, jaká část kódu se spustí na základě určitých podmínek. To zahrnuje základní příkazy jako `if` nebo `switch`, pomocí kterých mohou vytvářet rozhodovací logiku.
   - **Cykly** – Studenti by měli umět používat cykly (občas se můžeme setkat i s pojmem smyčky) jako `for` nebo `while`, které umožňují opakovaně spouštět určitou část kódu. Tyto konstrukce jsou klíčové pro úlohy, kde je potřeba zpracovávat větší množství dat nebo opakovaně provádět určité výpočty.
3. **Základní datové struktury** – Studenti by měli být seznámeni s několika základními datovými strukturami:
   (a) **Pole a seznamy (listy)** – Tyto struktury umožňují uchovávat více hodnot najednou a pracovat s nimi jako se skupinou. Pole a seznamy jsou nezbytné pro úlohy, kde se pracuje s většími množstvími dat, nebo kde se často používají opakující se hodnoty. Vzhledem k ukládání bitmapy je důležitá také znalost 2-rozměrných polí.
   (b) **Rozsahy (range)** – V některých jazycích se používá datová struktura zvaná range, což je posloupnost čísel v daném rozsahu, jinými slovy by se dalo říci, že se jedná o seznam (`list`) čísel v daném rozsahu. Range je užitečný zejména při práci s cykly, protože umožňuje snadno procházet určité intervaly čísel bez nutnosti manuálního výpočtu všech hodnot.

   Znalosti pokročilých datových struktur, jako jsou spojové seznamy, zásobníky, fronty, slovníky nebo stromy, nejsou nutné. Stejně tak se nepředpokládá nutná znalost časové či paměťové složitosti – avšak jejich znalost může být výhodou a přinést pochopení dalších znalostí.
4. **Funkce** – Studenti by měli rozumět konceptu funkcí a ideálně by měli umět základní funkce vytvářet. Funkce jsou samostatné bloky kódu, které lze opakovaně volat, a díky nim se program stává přehlednější a lépe strukturovaný. Pokud studenti ještě funkce sami tvořit neumí, měli by alespoň umět volat připravené funkce a chápat, jakým způsobem se jim předávají a vrací hodnoty.

3.4 Kotlin

V našich úlohách se budeme věnovat programování v jazyku Kotlin. Kotlin je moderní a efektivní jazyk, který je ideální pro vývoj aplikací i pro výuku základních a pokročilých programovacích konceptů. Zároveň také kombinuje jednoduchost s výkonností. Kotlin nabízí řadu vlastností, které ho činí užitečným nejen v praxi, ale i ve vzdělávacím prostředí[16]:

- **Konzistentní a přehledná syntaxe:** Kotlin minimalizuje množství opakujícího se kódu (tzv. bo-

ilerplate), což usnadňuje psaní, čtení i pochopení programů. Můžeme se tak lépe soustředit na samotné algoritmy a logiku, místo řešení složité syntaxe.

- **Bezpečnost při práci s null hodnotami:** Jedním z nejčastějších problémů v programování jsou tzv. „null pointer exceptions“, tedy chyby způsobené neošetřenými nulovými hodnotami. Kotlin tyto situace řeší zabudovanou null bezpečností, což podporuje osvojení lepších programovacích návyků a snižuje riziko chyb.
- **Propojení s jazykem Java:** Kotlin je plně kompatibilní s Javou, což znamená, že studenti mohou používat existující knihovny a frameworky psané v Javě. To poskytuje skvělou příležitost pro přechod od jednoduchých úloh k profesionálním projektům bez nutnosti učit se nový ekosystém od základů.
- **Moderní nástroje pro efektivní programování:** Kotlin nabízí několik užitečných nástrojů, které vám výrazně usnadní práci. *Rozšiřující funkce* umožňují přidávat nové vlastnosti do již existujících tříd bez jejich změny. *Korutiny* pak umožňují jednoduše provádět více úkolů současně, což je velmi užitečné pro aplikace, které potřebují běžet na pozadí, aniž by to zpomalilo hlavní část programu. *Datové třídy* zase automaticky generují základní funkce pro práci s daty, což šetří čas a usnadňuje manipulaci s nimi.
- **Multiplatformní vývoj:** S Kotlinem je možné vyvíjet aplikace pro více platforem (například Android, iOS, web). Tato schopnost je užitečná nejen v praxi, ale i pro pochopení konceptů sdíleného kódu a efektivního vývoje.

Kotlin je navržen tak, aby běžel na Java Virtual Machine (JVM), což je prostředí, ve kterém běží Java aplikace. Když napíšeme program v Kotlinu, tento kód se přeloží do bytekódu, který je následně spuštěn na JVM. JVM je zodpovědná za vykonávání bytekódu, což je strojový kód, který je nezávislý na platformě a umožňuje aplikacím běžet na různých typech zařízení. JVM je součástí Java Development Kit (JDK), případně pouze pro samotné spouštění programů v bytekódu stačí i Java Runtime Environment (JRE). JDK kromě JVM samotného obsahuje také potřebné nástroje pro kompilaci, vykonávání a ladění aplikací. JDK také poskytuje standardní knihovny a API, které Kotlin využívá pro různé funkce, jako jsou manipulace s textem, kolekcemi nebo vstupy/výstupy. Tato kompatibilita s Javou je jedním z hlavních důvodů, proč Kotlin vyžaduje JDK, protože Kotlin i Java sdílejí stejné prostředí a knihovny. To dává Kotlinu obrovskou flexibilitu a kompatibilitu s existujícím ekosystémem Java.

3.5 Použití materiálů ve výuce

Předkládané vzdělávací materiály jsou navrženy tak, aby sloužily jako flexibilní nástroj, použitelný jak při přímé výuce, tak při samostatném studiu. Materiály lze využít pro procvičování během výuky, nebo jako součást domácí přípravy. Úlohy jsou připravené tak, že obsahují nezbytnou teorii, takže učitel tak může studentům úlohy zadat jako samostatnou práci na procvičování a studenti tak mají vše co

potřebují. Tento přístup je umožněn díky kombinaci úloh s teoretickým základem, který studentům poskytuje nezbytné informace k jejich řešení.

Kapitola 1 obsahuje obecné teoretické základy grafiky. Tato kapitola studentům poskytne nezbytné znalosti, pokud se s grafickou oblastí zatím nesetkali a zároveň je kapitola postupně uvede do připravených výukových materiálů. Praktické části jsou rozděleny do kapitol 4 a 5. Kapitola 4 představuje konkrétní úlohu s podrobným popisem kroků k jejímu řešení a obsahuje vzorové řešení. Tu může učitel použít pro rozšíření výukových materiálů pro vlastní úlohy za použití ekosystému vytvořeného v této práci. Zároveň kapitola 4 také umožňuje studentům představit pohled na kompletní ekosystém a prohloubit porozumění o open-datových úlohách. Současně úloha též obsahuje vzorové řešení, takže studenti hned vidí, jak by jejich práce měla vypadat. Kapitola 5 se naopak zaměřuje na další úlohy, které jsou opatřeny zadáním a stručnou teorií, avšak bez ukázkových řešení. Kapitola 5 je koncipována tak, aby ji mohli studenti získat v elektronické či tištěné podobě a následně mohli řešit připravené úlohy. Dále z praktického hlediska budou potřebovat studenti také složku *students* z přílohy, kde je umístěn *generators.main.kts* a *judges.main.kts*. Tyto Kotlin skripty mohou použít pro generování vstupních dat a na kontrolu správnosti řešení. U každé úlohy bude uvedeno, s jakými parametry pro danou úlohu tyto programy spustit. Více informací o tom, jak pracovat s generátory a judge je uvedeno v sekce 3.7. Pokud studenti s programováním teprve začínají, může být také vhodné se studenty projít, jak si mohou připravit vývojové prostředí, případně jim poskytnout tyto informace (viz sekce 3.6 a sekce 3.4) k samostudiu.

Při práci s úlohami může být rovněž užitečná aplikace GeoGebra. Tento nástroj umožňuje zobrazit geometrické konstrukce, jako jsou body a úsečky, a dává studentům možnost snadno ověřit správnost své práce a porozumět výsledkům svého programování.

3.6 Příprava programovacího prostředí

3.6.1 Textový editor

Pro programování budeme potřebovat textový editor, pokud máme svůj oblíbený, tak můžeme použít ten. V opačném případě lze doporučit například Jetbrains Fleet, který nabízí pokročilou podporu pro Kotlin, nebo lze využít i Microsoft Visual Studio Code a nainstalovat si rozšíření pro Kotlin[1]. Alternativně lze také použít IDE *IntelliJ IDEA*, které však svým množstvím funkcí může být zahlcující.

Kromě textového editoru nebo IDE budeme potřebovat také JDK – pro spouštění generátorů a judge. V následující části je návod na instalaci. Pokud bychom chtěli programovat v jiném jazyce, pak můžeme instalaci Kotlinu vynechat a nainstalovat patřičné nástroje pro zvolený jazyk.

[1] Rozšíření lze nainstalovat ve VS Code otevřením „Palety příkazů“ (Ctrl + P, v angličtině „Quick Open“) a vložením příkazu `ext install fwcd.kotlin` s následným stisknutím enter ([17]).

3.6.2 Instalace na Windows

Instalace na Windows může být náročnější a proto si nastíníme několik možných variant.

Z dalších variant lze zmínit

- Použití WSL (instalace pomocí `wsl --install` a instalace podle instrukcí pro Linux (doporučená varianta)
- Instalace IntelliJ IDEA obsahující mimo jiné i Kotlin
- Stažení kompilátoru z oficiálního repozitáře na Githubu ([18]), extrahování na libovolné místo a v nastavení proměnných prostředí přidat cestu k adresáři bin[2] Kotlinu do systémových proměnných, aby bylo možné používat Kotlin z libovolného okna příkazového řádku
- Instalace OpenJDK a Kotlin kompilátorů přes chocolatey pomocí `choco install openjdk kotlinc`, zde bohužel ale není balíček Kotlinu udržovaný a je tak dostupný pouze ve starší verzi 1.5.20

Na základě zvolené varianty (starší verze Kotlinu) nemusí být dostupný příkaz kotlin, který budeme používat. Pokud příkaz není k dispozici, tak místo něj lze použít *kotlinc -script nazev-souboru.kts*.

3.6.3 Instalace na MacOS / Linux

Na MacOS či různých linuxových distribucích lze Kotlin a JDK nainstalovat přes jejich balíčkovací systémy, nebo použít SDKMAN!.

Instalace poté může vypadat takto, viz Zdrojový kód 1

```
1    curl -s "https://get.sdkman.io" | bash
2    sdk install java
3    sdk install kotlin
```

Zdrojový kód 1: Instalace JDK a Kotlinu pomocí SDKMAN!

V oficiální dokumentaci Kotlinu jsou případně zmíněny i další možnosti instalace.

Informace pro instalaci a spouštění vychází ze zdrojů: [16], [19], [20], [21], [22]

3.6.4 Použití Kotlinu

Odkaz na dokumentaci [16] doc api reference [23]

[2] Adresář bin je jeden z adresářů nacházející se ve stažené extrahované složce

Jednou z částí jazyka Kotlin je Kotlin Script (KTS), který umožňuje psát a spouštět kód jednoduše a bez potřeby vytváření kompletní aplikace. KTS je ideální pro rychlé experimentování, testování malých kousků kódu nebo automatizaci úloh, aniž by bylo nutné nastavovat složité projekty. S KTS můžeme psát skripty, které se přímo vykonají, což usnadňuje práci při prototypování, nebo řešení konkrétních problémů. Velmi jednoduchý program může vypadat například takto:

```
println("Ahoj světe!")
```

Pro spuštění skriptu použijeme příkaz *kotlinc -script*:

```
kotlinc -script ahoj.main.kts
```

Tento příkaz přeloží a vykoná obsah souboru přímo v konzoli. Také je důležité zmínit, že pro hlavní skript KTS platí, že musí za názvem mít příponu *.main.kts*

Užitečné odkazy

Velmi užitečný zdroj nejen při přípravě těchto úloh, ale i u jejich řešení je dokumentace[24] a API reference[23].

1. **Kotlin Dokumentace**: Kompletní dokumentace jazyka a jeho knihoven je k dispozici na Kotlin dokumentace.
2. **API Reference**: Pro detailní informace o Kotlin standardní knihovně můžeme použít Kotlin API Reference.

Organizace kódu

V Kotlinu je možné psát kód nejen do hlavního skriptu .kts, ale také do vedlejších .kt souborů, které lze následně importovat. Tato možnost je užitečná pro organizaci kódu, zejména pokud chceme opakovaně používat v různých skriptech stejné funkce nebo jiné opakující se části kódu. To může vypadat například takto:

```
// soubor utils.kt
fun sectiCisla(a: Int, b: Int) {
    println(a + b)
}
```

Tento soubor pak můžeme importovat do našeho hlavního skriptu .kts a používat jeho obsah takto:

```
// skript.main.kts
@file:Import("utils.kt")  // Importování souboru utils.kt
sectiCisla(2, 3)
```

V případě, že používáme editor JetBrains Fleet, tak musíme také v adresáři, který máme v editoru otevřený spustit příkaz *gradle init*, abychom zajistili správné fungování funkce Smart Mode[25].

Překompilování skriptů

Za zmínku také stojí vlastnost, která nemusí být na první pohled zřejmá, ale dokáže snadno zkomplikovat práci. Pokud v hlavním .kts souboru importujeme další .kt soubory, tak se při spuštění .kts automaticky zkompilují jak .kts soubor tak všechny používané .kt soubory. Zkompilované verze souborů se ukládají do cache a ke kompletnímu překompilování dojde pouze při změně v hlavním .kts souboru. Je tedy zapotřebí, myslet na to, že je vhodné vždy udělat změnu v hlavním .kts souboru, nebo smazat zkompilované soubory z cache, která se na linuxu nachází v */.cache/main.kts.compiled.cache/*. Smazáním této cache zajistíme, že při příštím spuštění se veškerý kód přeloží znovu, i když se změny týkají pouze importovaných souborů. Více informací lze nalézt na YouTrack[26], kde JetBrains trackuje chyby a úkoly pro další rozvoj Kotlinu.

Uložení výstupu do souboru

Výstup programu, který se běžně vypisuje do konzole terminálu, lze poměrně snadno přesměrovat a uložit do souboru. Tento soubor pak můžeme použít pro další práci – například pro vyhodnocení pomocí judge.

Jak je uvedeno na webu Kasiopea[27], výstup programu lze přesměrovat zhruba takto:

```
1  program.exe > reseni.txt              (Windows)
2  ./program > reseni.txt                (Linux)
3  python program.py > reseni.txt        (Python)
4  kotlin program.main.kts > reseni.txt  (Kotlin)
```

Tím se vyrobí soubor *reseni.txt* s výstupem programu.

Ve vývojových prostředích od JetBrains se dá přesměrování výstupu nastavit v menu Edit Configurations... na záložce Logs, kde je položka Save console output to, ve které se nastaví výstupní soubor.

3.7 Návod pro studenty

Pro práci s úlohami je nezbytné mít k dispozici složku *students* z přílohy. Nyní si ji podrobněji popíšeme:

- **Zdrojové kódy ke knihovně**: sloužka lib_sources obsahuje pomocnou knihovnu v podobě souboru bitmap.kt (větší detail o tom, co knihovna nabízí, lze nalézt v kapitola 7) a soubor *compile_lib.sh*, který zkompiluje bitmap.kt jako knihovnu a uloží ji do složky libs jako *lib.jar*
- **Složka libs**: Složka *libs* obsahuje zkompilované generátory, judge s řešením a pomocnou knihovnou zmíněnou v předchozím bodě

- **Soubor convert_bitmap_to_png.main.kts**: Kotlin skript, který převede námi specifikovanou bitmapu na PNG obrázek. Příklad spuštění: *kotlin convert_bitmap_to_png.main.kts bitmap_soubor nazev_obrazku.png*
- **Soubor generators.main.kts**: Kotlin skript, který vygeneruje náhodná vstupní data pro jakoukoliv úlohu. Argument pro tento skript obsahuje ID úlohy, pro kterou chceme generovat vstupní data, ID úlohy je vždy uvedeno v sekci „Generátor“ u jednotlivých úloh, typicky se jedná o ID jako 5_1 nebo 5_2. Příklad spuštění: `kotlin generators.main.kts 5_2`
- **Soubor judges.main.kts**: Kotlin skript, který vyhodnotí, zda je úloha vyřešena správně. Jako vstupní argumenty potřebuje zadat ID řešené úlohy (ID je uvedené v sekci „Judge“ u jednotlivých úloh) a poté názvy souborů s vygenerovanými vstupními daty a s výstupem, který chceme otestovat. Příklad spuštění: `kotlin judges.main.kts 5_2 generated_data bitmap_output`

Celkově tedy konzole terminálu při testování může vypadat takto:

```
> kotlin generators.main.kts 5_2 > generated_input_data
> kotlin program_resici_uloha.main.kts generated_input_data > bitmap_output
> kotlin judges.main.kts 5_2 generated_data bitmap_output
> kotlin convert_bitmap_to_png.main.kts bitmap_output nazev_obrazku.png
```

Kapitola 4

Fungování úlohy „Vodorovná čára“

Ve čtvrté kapitole podrobně rozebereme jednu z připravených úloh. Na začátku si ukážeme zadání úlohy následované obecným popisem, jak budeme úlohu řešit. Poté si prohlédneme, jak vypadá generátor a ukážeme si jeho použití. V této chvíli už budeme mít všechny informace k tomu, abychom mohli úlohu začít řešit. Pokud si však stále nebudeme jisti, jak úlohu řešit, nebo si nejsme jisti některou částí jejího řešení, můžeme nahlédnout do části „Krok za krokem“ – podsekce 4.1.5, kde je postupně popsáno jak úlohu naprogramovat. Následovat bude i vzorové řešení, které ukazuje, jak vypadá vyřešená úloha. V případě potřeby jej můžeme porovnat s naším řešením a najít příčinu případné chyby. Také můžeme porovnat, zda vzorové řešení není efektivnější než řešení studentů. Po jejím naprogramování si pomocí judge vyzkoušíme, zda je úloha správná. Ve zdrojovém kódu judge uvidíme, že program na základě vstupních dat porovná výstupní data ze studentova programu se svým řešením a vyhodnotí tak správnost úlohy.

4.1 Vodorovná čára

4.1.1 Zadání

Nechť máme obrázek o velikosti $w \times h$, kde w a h jsou kladná celá čísla. V daném obrázku chceme vykreslit vodorovnou čáru z bodu A do bodu B (pro vodorovnou čáru tedy platí $A_y = B_y$), body A a B jsou různé a jejich souřadnice jsou celočíselné. V této úloze budeme také předpokládat, že bod A je nalevo od bodu B (tedy platí $A_x < B_x$). Vykreslete obrázek, kde budou dané body propojené bílou čárou na černém pozadí.

Ve vstupu je na prvním řádku mezerou oddělené w a h. Na druhém a třetím řádku jsou x a y souřadnice bodů, které mají být propojeny čárou.

Vypište v námi specifikovaném BMP formátu (viz sekce 1.4). Data se vypíší do konzole terminálu.

Obtížnější varianta: Vyřešte úlohu, pokud neplatí podmínka, že bod A je nalevo od bodu B (tedy platí $0 < A_x < w$ a $0 < B_x < w$).

4.1.2 Ukázka

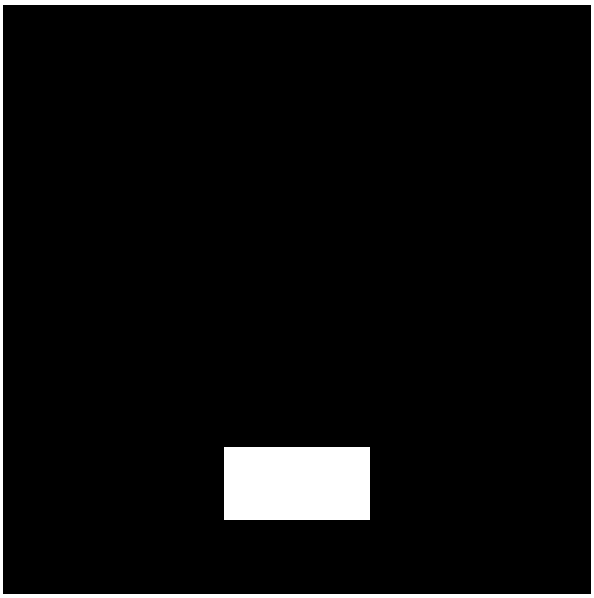
Vzor – vstupní data

```
8 8
3 1
4 1
```

Vzor – předpokládaný výstup

```
BMP,GREY
8 8
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 255 255 0 0 0
0 0 0 0 0 0 0 0
```

Vzor – předpokládaný výstup jako obrázek



Obrázek 4.1: Předpokládaný výstup jako obrázek pro úlohu „Kreslení čáry s anti-aliasingem“

4.1.3 Jak na to?

Začneme tím, že si vytvoříme bitmapu rozměrů $w \times h$, kde jednotlivé pixely budou mít bílou barvu. V této bitmapě poté projdeme pixely, skrze které prochází čára mezi zadanými body a na těchto pixelech změníme barvu na černou – to tedy v praxi znamená, že pomocí cyklu musíme projít všechny pixely od bodu A k bodu B. Ze zadání víme, že bod B je vždy napravo od bodu A – cyklus tedy vždy prochází souřadnice od A k bodu B (více viz obtížnější varianta). Na konci už bitmapu jen vypíšeme do konzole.

Jinými slovy musíme cyklem projít všechny souřadnice na ose X mezi bodem A a B (včetně souřadnic A_x a B_x). Na bodech, skrze které jsme prošli, musíme opět změnit barvu na černou. Vzhledem k tomu, že se jedná o vodorovnou čáru, tak souřadnice Y zůstává po celou dobu cyklu stejná (což mimo jiné vyplývá i ze zadání z části "platí $A_y = B_y$".

V případě obtížnější varianty je nutné si uvědomit, že body nebudou zadané tak, aby se čára vykreslovala od prvního bodu zleva doprava k druhému bodu[1]. Z toho důvodu musíme určit, který z bodů má menší X souřadnici (tedy který bod je více vlevo) – poté můžeme použít řešení z lehčí varianty a vytvořit cyklus, který začne v bodě více vlevo a postupně bude zvětšovat X souřadnici, dokud nedojdeme až k bodu více vpravo.

Poznámka: Je důležité podotknout, že možných způsobů, jak tuto úlohu vyřešit, existuje více, a každý z nich se může lišit v závislosti na zvoleném programovacím jazyku a konkrétním přístupu k implementaci.

4.1.4 Generátor zadání

Níže uvedený kód nám vygeneruje vstup odpovídající podmínkám zadání:

- Na řádcích 5 a 6 se vygenerují náhodné hodnoty pro w a h, což značí velikost bitmapy.
- Na řádku 8 se vygenerují dvě náhodná a vzájemně odlišná čísla, která se uloží jako a_x a b_x a použijí se při výpisu bodu A a bodu B.
- Na řádku 9 se vygeneruje náhodná souřadnice y, která se později použije jako souřadnice y pro vypsání bodu A i B.

Na řádcích 11 až 13 se poté už pouze hodnoty vypíší do konzole.

[1] Ve většině programovacích jazyků totiž u cyklu musíme určit inicializační hodnotu, na které cyklus začne a poté tuto hodnotu upravovat přičtením nebo odečtením, dokud není splněna ukončovací podmínka. Pokud ale nevíme, která ze souřadnic je větší, tak nevíme zda máme postupovat přičítáním nebo odečítáním.

```kotlin
package gen4

import kotlin.random.Random

fun generateTask4(minSize: Int, maxSize: Int): String {
    // vygenerování náhodných rozměrů čtvercové bitmapy
    val width = Random.nextInt(minSize, maxSize)
    val height = Random.nextInt(minSize, maxSize)

    // vygenerování náhodných x souřadnic bodů A a B
    val (a_x, b_x) = generateSequence { Random.nextInt(0, width) }.distinct().take(2).sorted().toList()
    // vygenerování náhodné y souřadnice totožné pro bod A i B
    val y = Random.nextInt(0, height)

    val result = StringBuilder().apply {
        appendLine("$width $height") // velikost bitmapy
        appendLine("$a_x $y") // bod A
        appendLine("$b_x $y") // bod B
    }
    return result.toString()
}
```

Zdrojový kód 2: Generátor 4.1

Generátor pro úlohu „Vodorovná čára“ spustíme ve složce `students` pomocí:

```
kotlin generators.main.kts 4
```

4.1.5 Řešení krok za krokem

1. Zpracujeme vstupní data a do proměnných si uložíme požadovanou šířku, výšku a souřadnice bodů A a B.
2. Vytvoříme si bitmapu, ve které budou všechny pixely černé. Nejjednodušší variantou je vytvořit 2rozměrné pole, kde každá buňka bude inicializována na 0 (černá).
3. **Volitelně**: Tento krok můžeme přeskočit, protože zadání nám zajišťuje, že tuto vlastnost vstupní data budou mít. Obecně je ale v programování vhodné vstupní data co nejvíce kontrolovat a ujistit se, zda jsou ve formátu, který požadujeme. Kontrolu můžeme provést porovnáním A_y a B_y – tím zkontrolujeme, zda vykreslovaná čára má být opravdu vodorovná.
4. **Tento krok je pouze v případě obtížné varianty**: Náš program předpokládá, že čáru budeme vykreslovat zleva doprava a proto potřebujeme zajistit, že bod A je nalevo od bodu B. To můžeme zajistit například tak, že do nové proměnné *min* uložíme minimální hodnotu z hodnot A_x a B_x. A obdobně do nové proměnné *max* uložíme maximální hodnotu. V dalším kroku poté použijeme místo A_x proměnnou *min* a místo B_x proměnnou *max*.

5. Pomocí cyklu projdeme všechny souřadnice od A_x (včetně) do B_x (včetně). Pro všechny pixely na iterované souřadnici mezi A_x a B_x se souřadnicí A_y, (resp. B_y, která má stejnou hodnotu) nastavíme hodnotu 255 (bílá).
6. Bitmapu vypíšeme.

4.1.6 Vzorové řešení

```
1  package solution_4
2
3  import java.io.File
4
5  fun solve4(inputFilename: String): String {
6      val file = File(inputFilename)
7      val linesList = file.readLines()
8
9      // ze vstupních dat uložíme do proměnné šířku a výšku bitmapy
10     val (width, height) = linesList[0].split(" ").map { it.toInt() }
11
12         // ze vstupních dat uložíme do proměnných souřadnice bodů A a B
13     val (a_x, a_y) = linesList[1].split(" ").map { it.toInt() }
14     val (b_x, b_y) = linesList[2].split(" ").map { it.toInt() }
15
16     // vytvoření prázdné bitmapy
17     val bitmap = Array(width) { Array<Int>(height) { 0 } }
18
19     // úprava barvy pixelů na souřadnicích čáry z bodu A do bodu B
20     for (i in a_x..b_x) {
21         bitmap[i][a_y] = 255
22     }
23
24     // vrácení bitmapy jako string, alternativně lze použít z knihovny funkci bitmapToString
25     val output = StringBuilder().apply {
26         appendLine("BMP,GREY")
27         appendLine("$width $height")
28         for (y in height - 1 downTo 0) {
29             for (x in 0 until width) {
30                 append("${bitmap[x][y]} ")
31             }
32             append("\n")
33         }
34     }
35     return output.toString()
36 }
37
```

Zdrojový kód 3: Vzorové řešení úlohy 4.1

Vzorové řešení – obtížnější varianta

```kotlin
package solution_4_hard

import kotlin.math.sign
import java.io.File

fun solve4Hard(inputFilename: String): String {
    val file = File(inputFilename)
    val linesList = file.readLines()

    // ze vstupních dat uložíme do proměnné šířku a výšku bitmapy
    val (width, height) = linesList[0].split(" ").map { it.toInt() }

        // ze vstupních dat uložíme do proměnných souřadnice bodů A a B
    val (a_x, a_y) = linesList[1].split(" ").map { it.toInt() }
    val (b_x, b_y) = linesList[2].split(" ").map { it.toInt() }

    // proměnná min obsahuje x souřadnici bodu, který je víc vlevo
    val min_x = minOf(a_x, b_x)
    // proměnná max obsahuje x souřadnici bodu, který je víc vpravo
    val max_x = maxOf(a_x, b_x)

    // vytvoření prázdné bitmapy
    val bitmap = Array(width) { Array<Int>(height) { 0 } }

    // úprava barvy pixelů na souřadnicích čáry z levého (menšího) bodu pravého (většího) bodu
    for (i in min_x..max_x) {
        bitmap[i][a_y] = 255
    }

    // vrácení bitmapy jako string, alternativně lze použít z knihovny funkci bitmapToString
    val output = StringBuilder().apply {
        appendLine("BMP,GREY")
        appendLine("$width $height")
        for (y in height - 1 downTo 0) {
            for (x in 0 until width) {
                append("${bitmap[x][y]} ")
            }
            append("\n")
        }
    }
    return output.toString()
}
```

Zdrojový kód 4: Vzorové řešení úlohy 4.1 – obtížnější varianta

4.1.7 Kontrolní judge

Judge pro úlohu „Vodorovná čára“ spustíme ve složce `students` pomocí:

```
kotlin judges.main.kts 4 data_z_generatoru vystup_z_programu_s_resenim
```

Kapitola 5

Zadání úloh

V páté kapitole si představíme jednotlivé úlohy. Každá úloha bude obsahovat zadání a postup, jak vytvořit řešení dané úlohy.

Vzorová řešení všech úloh nalezneme v kapitola 6

5.1 Vodorovná a svislá čára

5.1.1 Zadání

Nechť máme obrázek o velikosti $w \times h$, kde w a h jsou kladná celá čísla. V daném obrázku chceme vykreslit čáru z bodu A do bodu B. Tato čára může být vodorovná nebo svislá (pro vodorovnou čáru tedy platí $A_x = B_x$ a pro svislou $A_y = B_y$). Body A a B jsou různé. Vykreslete obrázek, kde budou dané body propojené bílou čárou na černém pozadí.

Ve vstupu je na prvním řádku mezerou oddělené w a h. Na druhém řádku vstupu jsou souřadnice bodu A ve formátu `x y`. Na třetím řádku vstupu jsou souřadnice bodu B ve formátu `x y`.

Vypište v námi specifikovaném BMP formátu (viz sekce 1.4). Data se vypíší do konzole terminálu.

5.1.2 Ukázka

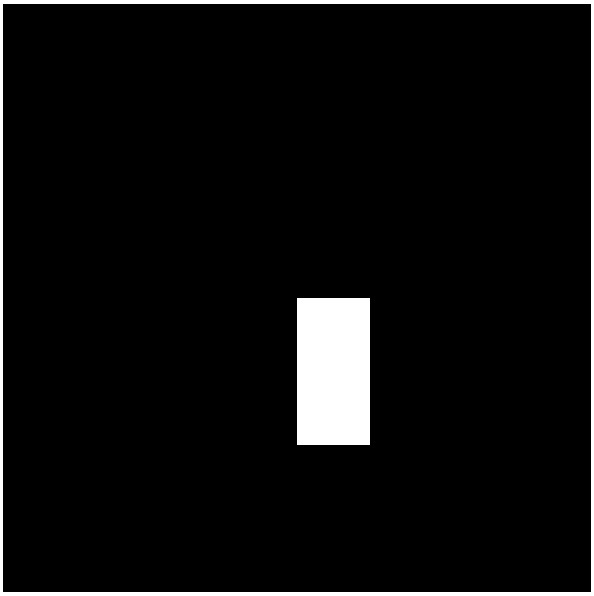
Vzor – vstupní data

```
8 8
4 2
4 3
```

Vzor – předpokládaný výstup

```
BMP,GREY
8 8
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 255 0 0 0
0 0 0 0 255 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

Vzor – předpokládaný výstup jako obrázek



Obrázek 5.1: Předpokládaný výstup jako obrázek pro úlohu „Vodorovná a svislá čára“

5.1.3 Jak na to?

Tuto úlohu lze rozdělit na dva případy – vykreslení vodorovné čáry a vykreslení svislé čáry. Z předchozí úlohy již umíme vykreslit vodorovnou čáru, takže tuto část můžeme aplikovat stejným způsobem i zde. Pro jednodušší použití je výhodné vytvořit funkci, která bude mít na starost právě vykreslení vodorovné čáry. Dále budeme potřebovat část programu, která vykreslí svislou čáru – to je velmi obdobný kód jako u vodorovné čáry a stačí nám tedy jej zkopírovat a upravit použití správných souřadnic. Opět se nám bude hodit vyčlenit tuto část programu do funkce pro vykreslení svislé čáry. Po spuštění programu zkontrolujeme vstupní data a zjistíme, zda se jedná o vodorovnou nebo svislou

čáru, poté na základě výsledku spustíme buď první nebo druhou funkci.

Alternativně lze úlohu řešit i tak, že si vytvoříme range pro souřadnice A_x až B_x, druhý range pro souřadnice A_y až B_y. Ze zadání lze odvodit, že pro svislou čáru bude $A_y = B_y$ a pro vodorovnou $A_x = B_x$. Z toho tedy vyplývá, že jeden range z těchto dvou range bude mít velikost právě 1 a bude v něm uložena buď pro oba body totožná souřadnice x nebo y. Oproti tomu druhý z range bude obsahovat seznam všech souřadnic mezi A_x až B_x nebo A_y až B_y. V úloze tak nepotřebujeme vědět, který range je který, protože velikost není určena podle toho, zda je čára vodorovná nebo svislá, ale podle vzdálenosti mezi danými souřadnicemi.

5.1.4 Generátor

Generátor pro úlohu „Vodorovná a svislá čára“ spustíme ve složce `students` pomocí:

```
kotlin generators.main.kts 5_1
```

5.1.5 Krok za krokem

Krok za krokem – řešení za pomoci funkcí

1. Zpracujeme vstupní data a do proměnných si uložíme požadovanou šířku, výšku a souřadnice bodů A a B,
2. Vytvoříme si bitmapu ve které všechny pixely nastavíme na barevnou hodnotu 0 (černá). Nejjednodušší variantou je, vytvořit 2rozměrné pole, kde každá buňka bude inicializována na 0.
3. Z úlohy v sekci 4.1 použijeme funkci pro vykreslení vodorovné čáry a nazveme ji například „drawHorizontalLine“. Funkci zkopírujeme a obdobně vytvoříme funkci „drawVerticalLine“, kde změníme použití x souřadnic na y souřadnice, místo osy X budeme iterovat na ose Y a při nastavování barevné hodnoty jednotlivých pixelů otočíme pořadí indexů.
4. Zjistíme, zda se jedná o vodorovnou nebo svislou čáru – to provedeme porovnáním x souřadnic bodů A a B. Pokud jsou x souřadnice shodné, vykreslíme čáru pomocí funkce *drawVerticalLine*. Naopak, pokud jsou shodné y souřadnice, pak vykreslíme čáru pomocí funkce *drawHorizontalLine*.
5. Bitmapu vypíšeme (pro výpis bitmapy můžeme použít funkci *printBitmap*, viz Zdrojový kód 25).

Krok za krokem – alternativní varianta

1. Zpracujeme vstupní data a do proměnných si uložíme požadovanou šířku, výšku a souřadnice bodů A a B.

2. Vytvoříme si bitmapu ve které budou všechny pixely mít barevnou hodnotu 0 (černá). Nejjednodušší variantou je, vytvořit 2rozměrné pole, kde každá buňka bude inicializována na 0.
3. Ze souřadnic spočítáme range přes který budeme iterovat. Range vytvoříme od x souřadnice levého bodu k x souřadnici pravého bodu. Obdobně vytvoříme další range od y souřadnice dolního bodu k y souřadnici horního bodu.
4. Cyklem iterujeme přes oba range. Pro každý sloupec s indexy z rangeX projdeme všechny řádky s indexy z rangeY a pro každý z pixelů nastavíme barevnou hodnotu na 255 (bílá).
5. Bitmapu vypíšeme (pro výpis bitmapy můžeme použít funkci *printBitmap*, viz Zdrojový kód 25).

5.1.6 Vzorové řešení

Vzorové řešení je v Zdrojový kód 5.

5.1.7 Kontrolní judge

Judge pro úlohu „Vodorovná a svislá čára“ spustíme ve složce `students` pomocí:

```
kotlin judges.main.kts 5_1_functions data_z_generatoru vystup_z_programu_s_resenim
```

Judge pro úlohu „Vodorovná a svislá čára – alternativní řešení“ spustíme ve složce `students` pomocí:

```
kotlin judges.main.kts 5_1_alternative data_z_generatoru vystup_z_programu_s_resenim
```

5.2 Čára v libovolném směru mezi 2 body

5.2.1 Zadání

Nechť máme obrázek o velikosti $w \times h$, kde w a h jsou kladná celá čísla. V daném obrázku chceme vykreslit čáru z bodu A do bodu B. Tato čára může být v libovolném směru. Body A a B jsou různé a jejich souřadnice jsou celočíselné. Vykreslete obrázek, kde budou dané body propojené bílou čárou na černém pozadí.

Ve vstupu je na prvním řádku mezerou oddělené w a h. Na druhém řádku vstupu jsou souřadnice bodu A ve formátu `x y`. Na třetím řádku vstupu jsou souřadnice bodu B ve formátu `x y`.

Vypište v námi specifikovaném BMP formátu (viz sekce 1.4). Data se vypíší do konzole terminálu.

5.2.2 Ukázka

Vzor – vstupní data

```
8 8
0 2
0 6
```

Vzor – předpokládaný výstup

```
BMP,GREY
8 8
0 0 0 0 0 0 0 0
255 0 0 0 0 0 0 0
255 0 0 0 0 0 0 0
255 0 0 0 0 0 0 0
255 0 0 0 0 0 0 0
255 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

Vzor – předpokládaný výstup jako obrázek

Obrázek 5.2: Předpokládaný výstup jako obrázek pro úlohu „Čára v libovolném směru mezi 2 body“

5.2.3 Jak na to?

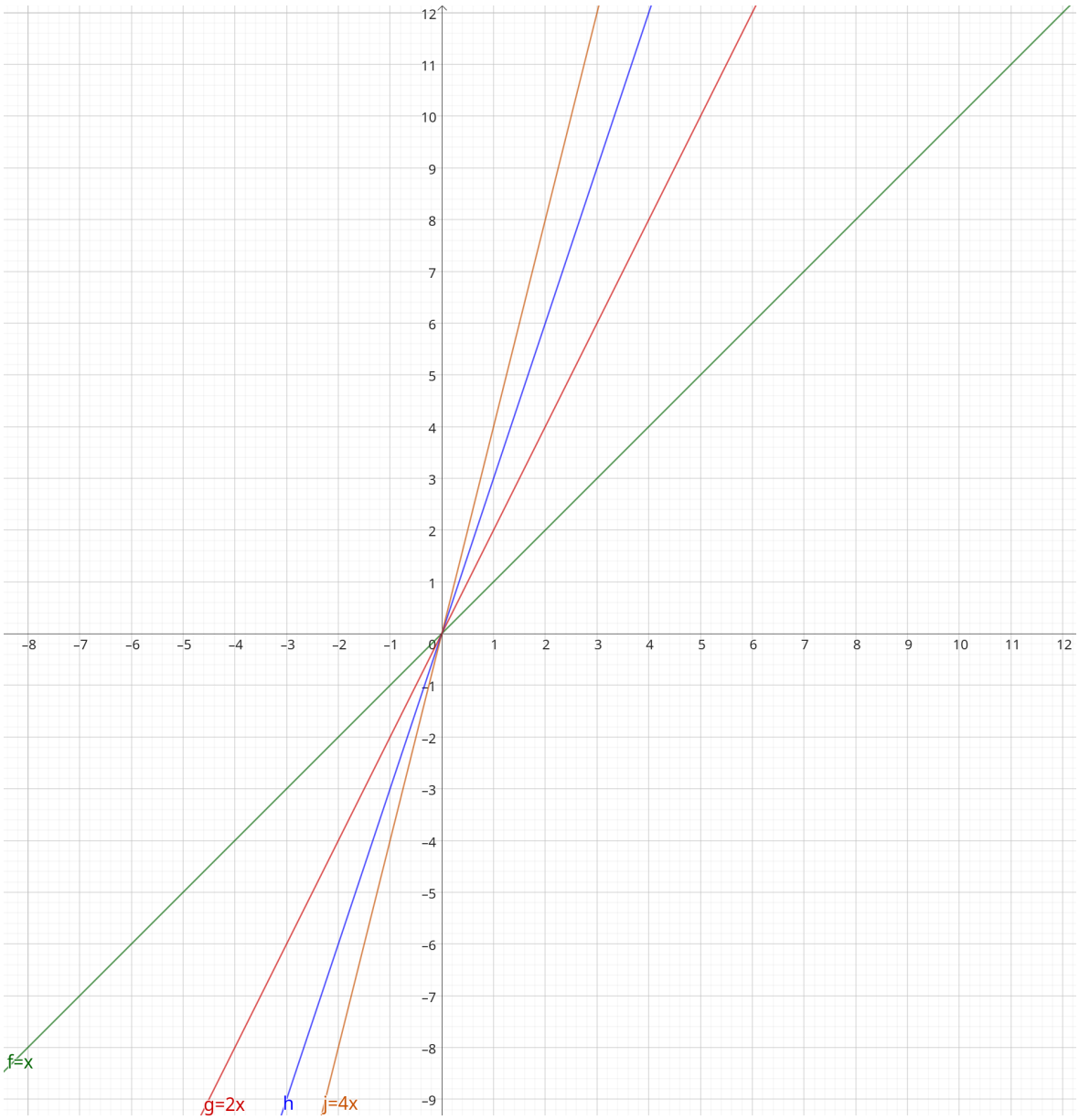
Na vykreslení přímky si ukážeme algoritmus DDA (digital differential algorithm), který je velmi jednoduchý. Budeme předpokládat, že body A a B mají celočíselné souřadnice, což plyne ze zadání. Pro použití algoritmu si můžeme představit lineární funkci, jejíž graf je přímka. Lineární funkce ve tvaru $y = kx + q$ bývá matematicky označena také jako směrnicový tvar přímky.

Pokud se podíváme na směrnicový tvar přímky trochu detailněji, tak jej lze popsat jako $y = kx + q$, kde k je směrnice (což je poměr hodnot směrového vektoru přímky) a q je hodnota definující posun přímky. V tomto vzorci lze také nahlédnout, že v případě kdy je $k = 0$, tak se jedná o vodorovnou čáru.

V případě úsečky můžeme směrový vektor definovat jako $\overrightarrow{u} = (y_2 - y_1, x_2 - x_1)$ a směrnici jako $k = (y_2 - y_1)/(x_2 - x_1)$. Jinými slovy se jedná o vektor skládající se ze vzdáleností Y souřadnic a X souřadnic našich dvou bodů.

Směrnice k vyjadřuje, o kolik se změní hodnota y (vertikální změna), když se hodnota x změní o 1 (horizontální změna). Uvedeme si několik příkladů (vizualizace viz Obrázek 5.3):

- Pokud $k = 1$, tak má přímka sklon 45°. To znamená, že při každé změně x o 1, změní se y o 1.
- Pokud $k = 2$, tak má přímka sklon 63.43°. To znamená, že při každé změně x o 1, změní se y o 2.
- Pokud $k = 3$, tak má přímka sklon 71.57°. To znamená, že při každé změně x o 1, změní se y o 3.
- Pokud $k = 4$, tak má přímka sklon 75.96°. To znamená, že při každé změně x o 1, změní se y o 4.



Obrázek 5.3: Graf přímek s různými směrnicemi k od 1 do 4. Přímky ukazují vliv směrnice k na sklon přímky. Vytvořeno v GeoGebra.

Algoritmus DDA[28] využívá právě změnu hodnoty y pro jednotlivé kroky na ose X. Definujme si tedy změnu pro jeden krok jako $dy = (y_2 - y_1)/(x_2 - x_1)$. Pro naši úsečku poté projdeme jednotlivé body (resp. pixely) a v každém kroku přičteme změnu pro jeden krok – dy do proměnné, která určuje hodnoty y v daném kroku. Nutnou podmínkou je, že musí být $x_1 < x_2$, tedy že se pohybujeme zleva doprava. Také musí platit, že směrnice je menší nebo rovno 1. Nevýhodou algoritmu DDA je, že dy může být i desetinné číslo a tedy je nutné v každém kroku provést zaokrouhlení na celé číslo. Na základě použitého datového typu může tedy vzniknout nepřesnost. Druhou nevýhodou algoritmu je operace dělení na začátku a operace zaokrouhlování v každém kroku, tyto operace patří k výpočetně náročnějším. Zejména u zařízení, jako jsou třeba různé 3D tiskárny nebo plottery, jsou k dispozici méně výkonné CPU a velmi často se využívá různých optimalizací – v takových případech se používá například Bresenhamův algoritmus. Za zmínku také stojí Hornerův algoritmus[29], který umí vykreslovat křivky, pokud je čára reprezentována jako polynom.

Zajímavost: Bresenhamův algoritmus

Bresenhamův algoritmus[30] se velice často používá na různých embedded zařízeních, kde nemáme k dispozici tolik výpočetního výkonu jako na PC a kde práce s desetinnými čísly může být náročnější (například není úplně praktické mít souřadnice s desetinnými hodnotami u 3D tiskárny, kde máme krokový motor, který pracuje s celými čísly). Naopak na takových zařízeních můžeme mnohem efektivněji využít násobení 2, což je ve dvojkové soustavě pouze bitový posun s přidáním 0, podobně jako když v desítkové soustavě násobíme 10 a také přidáme k číslu 0 na konec (tedy z 10 se stane 100). Kupříkladu tedy číslo 6 zapsané v binární soustavě jako 110 po vynásobení 2 vypadá v binární soustavě jako 1100, což je v desítkové soustavě 12.
Samotný algoritmus se odlišuje tím, že místo průběžného přičítání směrnice se v každém kroku vypočítá chybová odchylka oproti předchozímu kroku na základě následujícího vzorce:

$$E' = E + (dy)/(dx) <= 1/2$$

kde E' je chybová odchylka v aktuálním kroku a E je chybová odchylka z kroku předchozího. Na základě toho, zda je tato chybová odchylka menší nebo větší než 1/2, tak se pixel vykreslí na správné pozici. Postupnou úpravou (vynásobení $x2dx$ a odečtení dx) získáme vzorec:

$$dx(2E' - 1) = dx(2E - 1) + dy <= 0$$

a substitucí vytvoříme D, které označuje rozhodující proměnnou pro vykreslení pixelu.

$$D' = D + 2dy <= 0$$

Pokud je $D <= 0$, tak se průběžná hodnota y nemění. V opačném případě se průběžná hodnota y zvyšuje o 1 a z D se odečítá $2dx$, aby proměnná opět označovala rozhodující odchylku.

Při této příležitosti si můžeme také povšimnout, že všude máme pouze celá čísla a to znamená, že v tomto algoritmu nemůže nikdy docházet k žádným nepřesnostem.

5.2.4 Generátor zadání

Generátor pro úlohu „Čára v libovolném směru mezi 2 body“ spustíme ve složce `students` pomocí:

```
kotlin generators.main.kts 5_2
```

5.2.5 Krok za krokem

1. Zpracujeme vstupní data a do proměnných si uložíme požadovanou šířku, výšku a souřadnice bodů A a B.
2. Vytvoříme si bitmapu ve které budou všechny pixely mít barevnou hodnotu 0 (černá). Nejjednodušší variantou je, vytvořit 2rozměrné pole, kde každá buňka bude inicializována na 0.
3. Spočítáme pro obě osy rozdíl souřadnic bodů A a B – tedy jak daleko jsou od sebe body na jednotlivých osách.
4. Dále potřebujeme spočítat počet kroků – ten definuje, kolik pixelů bude potřeba projít při vykreslení čáry. Pro výpočet kroků nás ale zajímá pouze jejich vzdálenost – tedy absolutní hodnota rozdílu souřadnic bodů A a B. Výsledný počet kroků tedy určíme jako maximální hodnotu z absolutní hodnoty těchto rozdílů. To znamená, že čára bude vykreslena v počtu kroků rovnoměrně podél té osy, kde je větší vzdálenost.
5. V další fázi potřebujeme spočítat přírůstky na každém kroku. Tyto přírůstky určují, jak velký bude posun na každé ose při každém kroku, jinými slovy, o kolik se souřadnice změní na každé ose během každého kroku. Přírůstek počítáme pro každou osu a vypočítáme jej jako podíl vzdálenosti bodů A a B na příslušné ose a počet kroků (step). Výsledný přírůstek na ose X spočítáme jako $\text{diffX} = \frac{\text{lengthX}}{\text{step}}$
6. Vytvoříme si proměnné pro průběžné souřadnice x, y. Tyto proměnné nám budou říkat, s jakým pixelem zrovna pracujeme. Jejich inicializační hodnotou budou souřadnice levého dolního bodu.
7. Cyklem iterujeme od 0 do počtu kroků (step) a v každé iteraci zaokrouhlíme průběžné souřadnice x, y a na jejich pixelu nastavíme barevnou hodnotu na 255 (bílá), poté ještě zvýšíme průběžné souřadnice x, y o přírůstky os X a Y.
8. Bitmapu vypíšeme (pro výpis bitmapy můžeme použít funkci *printBitmap*, viz Zdrojový kód 25).

Pokud bychom vykreslovali čáru z bodu $(2, 3)$ do bodu $(15, 8)$, iterace cyklem bude probíhat podle hodnot uvedených v Tabulka 5.1.

Krok i	x	y	**Zaokrouhlené** (x, y)	**Popis**
0	2.0	3.0	(2, 3)	Počáteční bod
1	3.0	3.3846	(3, 3)	Další bod
2	4.0	3.7692	(4, 4)	Další bod
3	5.0	4.1538	(5, 4)	Další bod
4	6.0	4.5384	(6, 5)	Další bod
5	7.0	4.9230	(7, 5)	Další bod
6	8.0	5.3076	(8, 5)	Další bod
7	9.0	5.6922	(9, 6)	Další bod
8	10.0	6.0768	(10, 6)	Další bod
9	11.0	6.4614	(11, 6)	Další bod
10	12.0	6.8460	(12, 7)	Další bod
11	13.0	7.2306	(13, 7)	Další bod
12	14.0	7.6152	(14, 8)	Další bod
13	15.0	8.0	(15, 8)	Konec čáry

Tabulka 5.1: Iterace při vykreslování čáry z bodu $(2, 3)$ do bodu $(15, 8)$ pomocí DDA algoritmu

5.2.6 Vzorové řešení

Vzorové řešení je v Zdrojový kód 7

5.2.7 Kontrolní judge

Judge pro úlohu „Čára v libovolném směru mezi 2 body“ spustíme ve složce `students` pomocí:

```
kotlin judges.main.kts 5_2 data_z_generatoru vystup_z_programu_s_resenim
```

5.3 Obdélník

5.3.1 Zadání

Nechť máme obrázek o velikosti $w \times h$, kde w a h jsou kladná celá čísla. V daném obrázku chceme vykreslit čáry propojující 4 body ($ABCD$), tak aby vznikl bílý obdélník na černém pozadí. Vygenerované body jsou různé a jejich souřadnice jsou celočíselné. Bod A je vlevo dole, bod B je vpravo dole, bod C je vpravo nahoře a bod D je vlevo nahoře (viz Obrázek 5.4). Vykreslete obrázek, kde bude na černém pozadí obdélník se stranami bílé barvy.

Obrázek 5.4: Obdélník s body A, B, C, D vykreslený v GeoGebra

Ve vstupu je na prvním řádku mezerou oddělené w a h. Na druhém řádku vstupu jsou souřadnice bodu A ve formátu `x y`. Na třetím řádku vstupu jsou souřadnice bodu B ve formátu `x y`. Na čtvrtém řádku vstupu jsou souřadnice bodu C ve formátu `x y`. Na pátém řádku vstupu jsou souřadnice bodu D ve formátu `x y`.

Vypište v námi specifikovaném BMP formátu (viz sekce 1.4). Data se vypíší do konzole terminálu.

5.3.2 Ukázka

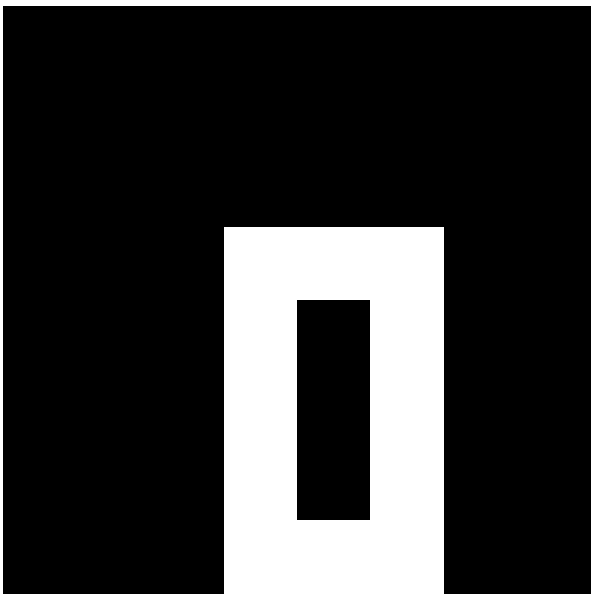
Vzor – vstupní data

```
8 8
3 0
5 0
5 4
3 4
```

Vzor – předpokládaný výstup

```
BMP,GREY
8 8
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 255 255 255 0 0
0 0 0 255 0 255 0 0
0 0 0 255 0 255 0 0
0 0 0 255 0 255 0 0
0 0 0 255 255 255 0 0
```

Vzor – předpokládaný výstup jako obrázek



Obrázek 5.5: Předpokládaný výstup jako obrázek pro úlohu „Obdélník“

5.3.3 Jak na to?

Použijeme řešení úlohy „Vodorovná a svislá čára“ (sekce 5.1) pro vykreslení jednotlivých stran obdélníku.

5.3.4 Generátor zadání

Generátor pro úlohu „Obdélník“ spustíme ve složce `students` pomocí:

```
kotlin generators.main.kts 5_3
```

5.3.5 Krok za krokem

1. Zpracujeme vstupní data a do proměnných si uložíme požadovanou šířku, výšku a souřadnice bodů A, B, C a D.
2. Vytvoříme si bitmapu ve které budou všechny pixely mít barevnou hodnotu 0 (černá). Nejjednodušší variantou je, vytvořit 2rozměrné pole, kde každá buňka bude inicializována na 0.
3. Vykreslíme čáry pomocí funkcí *drawVerticalLine* a *drawHorizontalLine* z úlohy „Vodorovná a svislá čára“ v sekce 5.1, případně můžeme čáry vykrkeslit i pomocí funkce *drawLine* z úlohy „Čára v libovolném směru mezi 2 body“ v sekce 5.2.
4. Jednotlivé čáry k vykreslení tedy jsou
   - mezi body A a B (funkce *drawHorizontalLine* nebo *drawLine*)
   - mezi body B a C (funkce *drawVerticalLine* nebo *drawLine*)
   - mezi body C a D (funkce *drawHorizontalLine* nebo *drawLine*)
   - mezi body D a A (funkce *drawVerticalLine* nebo *drawLine*)
5. Bitmapu vypíšeme (pro výpis bitmapy můžeme použít funkci *printBitmap*, viz Zdrojový kód 25).

5.3.6 Kontrolní judge

Judge pro úlohu „Obdélník“ spustíme ve složce `students` pomocí:

```
kotlin judges.main.kts 5_3 data_z_generatoru vystup_z_programu_s_resenim
```

5.4 Vyplněný obdélník

5.4.1 Zadání

Nechť máme obrázek o velikosti $w \times h$, kde w a h jsou kladná celá čísla. V daném obrázku chceme vykreslit vyplněný obdélník. V zadání máme pouze bod A a C. Vygenerované body jsou různé a jejich souřadnice jsou celočíselné. Bod A je levý dolní roh obdélníku a bod C je pravý horní roh obdélníku. Vykreslete obrázek, kde bude vyplněný bílý obdélník na černém pozadí.

Ve vstupu je na prvním řádku mezerou oddělené w a h. Na druhém řádku vstupu jsou souřadnice bodu A ve formátu `x y`. Na třetím řádku vstupu jsou souřadnice bodu C ve formátu `x y`.

Vypište v námi specifikovaném BMP formátu (viz sekce 1.4). Data se vypíší do konzole terminálu.

5.4.2 Ukázka

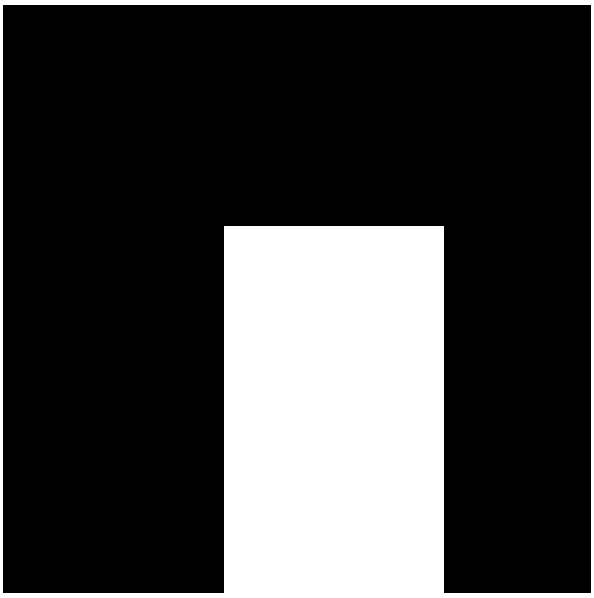
Vzor – vstupní data

```
8 8
3 0
5 4
```

Vzor – předpokládaný výstup

```
BMP,GREY
8 8
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 255 255 255 0 0
0 0 0 255 255 255 0 0
0 0 0 255 255 255 0 0
0 0 0 255 255 255 0 0
0 0 0 255 255 255 0 0
```

Vzor – předpokládaný výstup jako obrázek



Obrázek 5.6: Předpokládaný výstup jako obrázek pro úlohu „Vyplněný obdélník“

5.4.3 Jak na to?

U této úlohy se vrátíme k úloze v sekce 5.1. Pokud jsme u minulé úlohy postupovali podle alternativního řešení, vytvořili jsme range na ose X a range na ose Y. Protože jsme vykreslovali pouze čáru, věděli jsme, že jedna ze souřadnic bodů A a B musí být stejná. Díky tomu měl jeden z těchto range vždy velikost 1, což znamenalo, že rozsah hodnot byl omezený pouze na jednu pevnou souřadnici.

Pokud by však tato podmínka neplatila, tedy pokud by oba range měly různé velikosti větší než 1, situace by se výrazně změnila. V takovém případě bychom během průchodu jednotlivými řádky museli zároveň projít též všemi sloupci. Tento postup by vedl k vytvoření vyplněného obdélníku, který by zahrnoval všechny body mezi definovanými range na ose X a Y. Výsledný obdélník by pak byl definován dvěma body, které by ležely na společné úhlopříčce, což je charakteristický rys tohoto postupu.

5.4.4 Generátor zadání

Generátor pro úlohu „Vyplněný obdélník“ spustíme ve složce `students` pomocí:

```
kotlin generators.main.kts 5_4
```

5.4.5 Krok za krokem

- Zpracujeme vstupní data a do proměnných si uložíme požadovanou šířku, výšku a souřadnice bodů A a B.
- Vytvoříme si bitmapu, ve které budou všechny pixely mít barevnou hodnotu 0 (černá). Nejjednodušší variantou je, vytvořit 2rozměrné pole, kde každá buňka bude inicializována na 0.
- Vytvoříme rangeY, který bude definovat všechny řádky mezi Y souřadnicemi dvou bodů.
- Vytvoříme rangeX, který bude definovat všechny sloupce mezi X souřadnicemi dvou bodů.
- Pomocí cyklu poté iterujeme celým rangeY a v rámci každé této iterace také iterujeme přes celý rangeX, kde jednotlivým pixelům změníme barevnou hodnotu na 255 (bílá).
- Bitmapu vypíšeme (pro výpis bitmapy můžeme použít funkci *printBitmap*, viz Zdrojový kód 25).

5.4.6 Kontrolní judge

Judge pro úlohu „Vyplněný obdélník“ spustíme ve složce `students` pomocí:

```
kotlin judges.main.kts 5_4 data_z_generatoru vystup_z_programu_s_resenim
```

5.5 Mnohoúhelník

5.5.1 Zadání

Na vstupu dostaneme n bodů a z nich vykreslíme n-úhelník (příklad: pro $n = 5$ budeme mít čtyřúhelník, jehož vrcholy jsou body A, B, C, D, E a vykreslíme úsečky AB,BC,CD,DE a EA). Jednotlivé strany tohoto n-úhelníku mohou svírat libovolné úhly. Pro vygenerovaná vstupní data platí, že $n >= 3$.

Nechť máme obrázek o velikosti $w \times h$, kde w a h jsou kladná celá čísla. V daném obrázku chceme vykreslit n-úhelník. Vygenerované body jsou různé a jejich souřadnice jsou celočíselné. Vykreslete obrázek, kde budou dané body propojené čárou.

Ve vstupu je na prvním řádku mezerou oddělené w a h. Na druhém řádku je počet bodů pro n-úhelník. Na 3. až $n + 3$. řádku jsou x a y souřadnice bodů, které mají být propojeny čárou.

Vypište v námi specifikovaném BMP formátu (viz sekce 1.4). Data se vypíší do konzole terminálu.

Zajímavost

Při řešení této úlohy je také zajímavé zamyslet se nad tím, jak se vzhled mnohoúhelníku mění se zvyšujícím se počtem stran n. Pokud jde o pravidelný mnohoúhelník, jehož všechny vrcholy leží na společné kružnici, bude se při rostoucím n stále více podobat samotné kružnici. S rostoucím počtem stran se totiž zmenšují rozdíly mezi jednotlivými úsečkami tvořícími obvod mnohoúhelníku. Při dostatečně velkém n přestane lidské oko tyto rozdíly rozlišovat. To znamená, že od určité hodnoty n již není vizuálně patrný rozdíl mezi mnohoúhelníkem o n stranách a o $n + 1$ stranách, což vytváří iluzi dokonale hladkého kruhu.

5.5.2 Ukázka

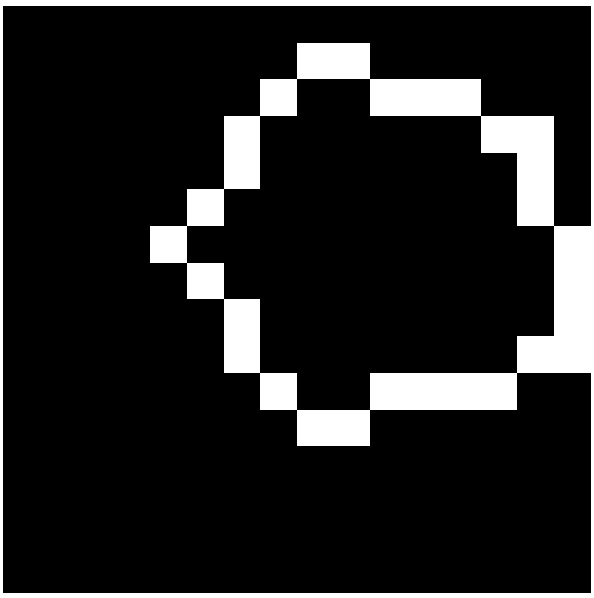
Vzor – vstupní data

```
16 16
5
14 12
8 14
4 9
8 4
15 6
```

Vzor – předpokládaný výstup

```
BMP,GREY
16 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 255 255 0 0 0 0 0 0
0 0 0 0 0 0 0 255 0 0 255 255 255 0 0 0
0 0 0 0 0 0 255 0 0 0 0 0 0 255 255 0
0 0 0 0 0 0 255 0 0 0 0 0 0 0 255 0
0 0 0 0 0 255 0 0 0 0 0 0 0 0 255 0
0 0 0 0 255 0 0 0 0 0 0 0 0 0 0 255
0 0 0 0 0 255 0 0 0 0 0 0 0 0 0 255
0 0 0 0 0 0 255 0 0 0 0 0 0 0 0 255
0 0 0 0 0 0 255 0 0 0 0 0 0 0 255 255
0 0 0 0 0 0 0 255 0 0 255 255 255 255 0 0
0 0 0 0 0 0 0 0 255 255 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Vzor – předpokládaný výstup jako obrázek



Obrázek 5.7: Předpokládaný výstup jako obrázek pro úlohu „Mnohoúhelník“

5.5.3 Jak na to?

Využijeme znalosti toho, že umíme vykreslit čáru mezi dvěma různými body. Cyklem iterujeme přes zadané body (kromě posledního) a postupně každý propojíme čárou s dalším. Poslední bod poté propojíme s prvním bodem.

5.5.4 Generátor zadání

Generátor pro úlohu „Mnohoúhelník“ spustíme ve složce `students` pomocí:

```
kotlin generators.main.kts 5_5
```

5.5.5 Krok za krokem

1. Zpracujeme vstupní data a do proměnných si uložíme požadovanou šířku, výšku a počet bodů n. Následně si uložíme do seznamu i n bodů na dalších řádcích.
2. Vytvoříme si bitmapu ve které budou všechny pixely mít barevnou hodnotu 0 (černá). Nejjednodušší variantou je, vytvořit 2rozměrné pole, kde každá buňka bude inicializována na 0.
3. Cyklem iterujeme (například s proměnnou `i`) od n do $n-1$ a pomocí funkce `vykresliCaru` vykreslíme čáru mezi aktuálním a následujícím bodem ze seznamu – tedy mezi body ze seznamu s indexem i a $i+1$.
4. Mnohoúhelník uzavřeme vykreslením čáry mezi prvním a posledním bodem ze seznamu.
5. Bitmapu vypíšeme (pro výpis bitmapy můžeme použít funkci *printBitmap*, viz Zdrojový kód 25).

5.5.6 Kontrolní judge

Judge pro úlohu „Mnohoúhelník“ spustíme ve složce `students` pomocí:

```
kotlin judges.main.kts 5_5 data_z_generatoru vystup_z_programu_s_resenim
```

5.6 Kreslení čáry s anti-aliasingem

5.6.1 Anti-aliasing v DDA algoritmu

Bresenhamův algoritmus umí čáru vykreslit přesně a velmi rychle, ale bohužel neumí čáru vykreslit s anti-aliasingem. Jeden z nejjednodušších způsobů jak vykreslit čáru s anti-aliasingem, je použití rozšířeného DDA algoritmu. Tímto rozšířením dokážeme zlepšit vzhled čáry vykreslené na mřížce pixelů, aby nevypadala „zubatě“. Anti-aliasing vytváří iluzi hladké čáry tím, že jemně zmírňuje přechody mezi barvou čáry a pozadím. Čára pak vypadá jako plynulejší a více přirozená. Takto rozšířený DDA algoritmus se nazývá Xiaolin Wuův algoritmus[31]. Pro řešení naší úlohy si ale ukážeme praktičtější přístup – super-sampling (podsekce 5.6.2).

Zajímavost: Xiaolin Wuův algoritmus podrobněji

U běžného DDA algoritmu vybíráme vždy jeden pixel nejblíže ideální čáře, který na obrazovku vykreslíme jako „čáru“. Pokud ale přidáme anti-aliasing, kromě hlavního pixelu změníme barvu i okolních pixelů, které jsou blízko čáry. Tyto pixely obarvíme světlejší verzí barvy čáry, což vytváří plynulý přechod.

Krok 1: Výběr okolních pixelů

Anti-aliasing u Xiaolin Wuova algoritmu zkoumá dva hlavní sousední pixely vedle každého bodu na čáře:

- Hlavní pixel, který leží na čáře nebo je jí nejblíže.
- Sousední pixely, které se nacházejí nad a pod primárním pixelem (u čar s menším sklonem) nebo vlevo a vpravo (u čar s větším sklonem).

V algoritmu Xiaolin Wu se vybírají sousední pixely na základě sklonu čáry. Pokud má čára větší sklon (je více svislá), sousední pixely, které přispívají k anti-aliasingu, se nacházejí vlevo a vpravo od primárního pixelu. Naopak, pokud má čára menší sklon (je více vodorovná), sousední pixely leží nad a pod primárním pixelem. Tento výběr sousedních pixelů zajišťuje, že jsou správně zahrnuty pixely nejblíže ideální čáře, což vytváří hladší přechod mezi čarou a pozadím.

Krok 2: Určení intenzity barvy

Intenzita (nebo průhlednost) každého sousedního pixelu závisí na tom, jak daleko je od ideální čáry. Čím blíže je pixel k čáře, tím tmavší bude – tedy intenzivnější barvou. Tento efekt je nejsilnější u hlavního pixelu a zeslabuje se u sousedních.

Lze si tedy představit, že intenzita je jako „síla“ barvy. Hlavní pixel je nejtmavší a pixely kolem budou stále světlejší.

Poté podle toho, zda:

- je sousední pixel velmi blízko čáře, může mít intenzitu třeba 70 % barvy čáry.

- je pixel trochu dál, může mít jen 30 % nebo méně.

Pokud si představíme příklad, kdy máme černou čáru na bílém pozadí, tak hlavní pixel bude černý. Sousední pixely, které jsou blízko čáře, budou světle šedé. Tím docílíme plynulého přechodu mezi čárou a pozadím.
Tímto postupem vytvoříme iluzi hladké čáry s anti-aliasingem.

5.6.2 Super-sampling

Další z efektivních možností, jak dosáhnout efektu anti-aliasingu, je kreslit čáru (nebo i jiné tvary) ve vyšším rozlišení a poté ji zmenšit do původního rozlišení. Tato metoda, známá jako super-sampling[32][33], využívá jemnější detaily ve vyšším rozlišení, takže po zmenšení do původního rozlišení vznikne iluze plynulejších hran.

Postup se super-samplingem:

1. Zvýšení rozlišení: Představte si, že máte cílový obraz 100×100 pixelů. Při super-samplingu jej zvětšíte, například 2x nebo 4x, tedy na 200×200 nebo 400×400 pixelů.
2. Vykreslení tvarů ve vyšším rozlišení: Do této větší mřížky vykreslíte čáry nebo jiné tvary.
3. Zmenšení do původního rozlišení: Výsledný obraz zmenšíte zpět do cílového rozlišení, čímž se jemnější detaily sloučí a vzniknou přechody, které vytvářejí iluzi hladkých hran.

Volba 2x nebo 4x vyššího rozlišení je běžná, protože jsou to mocniny čísla 2, což se v digitálním zpracování obrazu často používá kvůli efektivnímu výpočtu a zpracování. Při hodnotě 2× má každý cílový pixel detail ze čtyř pixelů v super-samplovaném rozlišení, a při hodnotě 4× získává každý cílový pixel průměrnou informaci ze 16 pixelů. Tato mocninná zvětšení umožňují lepší kontrolu nad detaily i při zmenšení, zatímco vyšší faktory (např. 8×) by výrazně zpomalily vykreslování a spotřebovaly více paměti.

Super-sampling je obzvlášť užitečný, pokud vykreslujeme více čar a tvarů nebo pokud se tvary protínají, protože vyhlazuje přechody mezi všemi částmi obrazu. Obraz zmenšený ze super-samplovaného rozlišení poskytuje často plynulejší hrany než metody, které řeší anti-aliasing u jednotlivých pixelů v základním rozlišení.

Super-sampling však vyžaduje více paměti a výpočetního výkonu, protože obraz je vykreslován ve větším rozlišení a poté zmenšován. S vyšším rozlišením také roste paměťová náročnost, což může zpomalit vykreslování na větších plochách.

Naopak Xiaolin Wuův algoritmus s anti-aliasingem mění jen pár sousedních pixelů kolem čáry, takže spotřebovává méně paměti a běží rychleji. Pro vykreslení jednoduché čáry je tento přístup efektivnější než super-sampling, ale vyhlazení nemusí být tak přesné jako při super-smaplingu (zejména u tenkých

čar nebo jiných složitějších tvarů). Navíc pokud se čáry a tvary protínají nebo je jich mnoho, metoda anti-aliasingu přímo v algoritmu Xiaolina Wua nezajistí tak plynulé přechody jako super-sampling.

5.6.3 Zadání

Na vstupu dostaneme n dvojic bodů. Každá dvojice obsahuje 2 různé body.

Nechť máme obrázek o velikosti $w \times h$, kde w a h jsou kladná celá čísla. V daném obrázku chceme vykreslit n čar definovaných jednotlivými dvojicemi. Po jejich vykreslení chceme na obrázek aplikovat anti-aliasing pomocí super-samplingu.

Ve vstupu je na prvním řádku mezerou oddělené w a h. Druhý řádek obsahuje super-sampling faktor, což je celé číslo f a platí, že $f >= 2$. Třetí řádek obsahuje celé číslo n, které určuje počet dvojic bodů na 4. až $n + 4$. řádku. 4. až $n + 4$. řádek obsahuje čtyři celá čísla ve formátu `a b,c d`. Čísla a a b představují souřadnice x a y prvního bodu dvojice, zatímco čísla c a d představují souřadnice x a y druhého bodu.

Vypište v námi specifikovaném BMP formátu (viz sekce 1.4). Data se vypíší do konzole terminálu.

5.6.4 Ukázka

Vzor – vstupní data

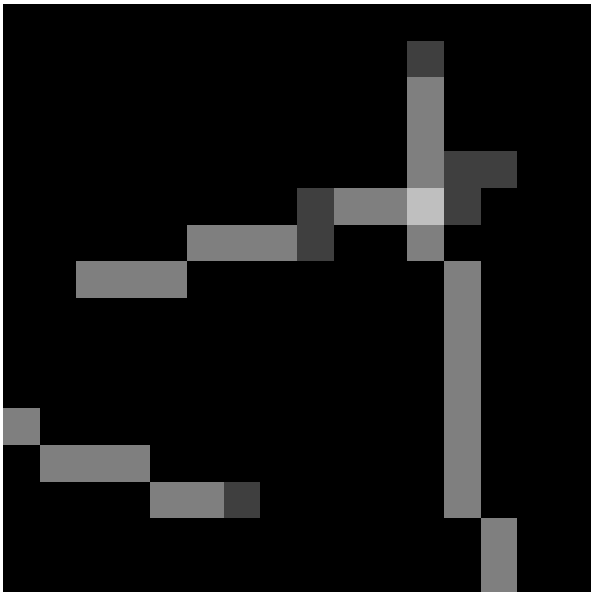
```
16 16
2
3
0 4,6 2
13 11,2 8
11 14,13 0
```

Vzor – předpokládaný výstup

```
BMP,GREY
16 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 63 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 127 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 127 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 127 63 63 0 0
0 0 0 0 0 0 0 0 63 127 127 191 63 0 0 0
```

```
0 0 0 0 0 127 127 127 63 0 0 127 0 0 0 0
0 0 127 127 127 0 0 0 0 0 0 0 127 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 127 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 127 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 127 0 0 0
127 0 0 0 0 0 0 0 0 0 0 0 127 0 0 0
0 127 127 127 0 0 0 0 0 0 0 0 127 0 0 0
0 0 0 0 127 127 63 0 0 0 0 0 127 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 127 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 127 0 0
```

Vzor – předpokládaný výstup jako obrázek



Obrázek 5.8: Předpokládaný výstup jako obrázek pro úlohu „Kreslení čáry s anti-aliasingem“

5.6.5 Jak na to?

U této úlohy budeme postupovat velmi podobně jako u úlohy Mnohoúhelník(viz sekce 5.5), pouze nebudeme vykreslovat čáry mezi po sobě jdoucími body, ale mezi body ze seznamu dvojic. Je však důležité si uvědomit, že po aplikování super-samplingu se nám obrázek zmenší 2x (popřípadě jinou hodnotou podle faktoru). Abychom tedy splnili zadání, musíme body vykreslit do obrázku 2x většího a následným zmenšením nám poté vznikne obrázek s čárami, které mají anti-aliasing a zároveň jsou vykreslené mezi zadanými body.

5.6.6 Generátor

Generátor pro úlohu „Kreslení čáry s anti-aliasingem“ spustíme ve složce `students` pomocí:

```
kotlin generators.main.kts 5_6
```

5.6.7 Krok za krokem

1. Zpracujeme vstupní data a do proměnných si uložíme požadovanou šířku, výšku, super-sampling faktor[1], počet dvojic bodů a jednotlivé dvojice bodů. Protože budeme čáry mezi body vykreslovat do bitmapy zvětšené *super-sampling faktorem*, tak je vhodné souřadnice bodů už během ukládání vynásobit hodnotou *super-sampling faktoru*.
2. Vytvoříme si bitmapu, jejíž šířka i výška je násobená hodnotou *super-sampling faktoru*. Pokud máme na vstupu šířku a výšku 10, *super-sampling faktor* 2, tak rozměry bitmapy budou 20x20. Do této bitmapy budeme vykreslovat čáry ve zvýšeném rozlišení. Všechny pixely budou mít barevnou hodnotu 0 (černá). Nejjednodušší variantou je, vytvořit 2rozměrné pole, kde každá buňka bude inicializována na 0.
3. Do zvětšené bitmapy pro všechny dvojice bodů vykreslíme čáru bílé barvy mezi prvním a druhým bodem v jednotlivých dvojicích (pro vykreslení čáry mezi 2 body můžeme použít funkci *drawLine* z úlohy „Čára v libovolném směru mezi 2 body“ v sekce 5.2.)
4. Vytvoříme si další bitmapu, tentokrát o velikosti zadané na vstupu. Do této bitmapy budeme překreslovat zvětšenou bitmapu s efektem anti-aliasingu. Všechny pixely budou mít barevnou hodnotu 0 (černá). Nejjednodušší variantou je, vytvořit 2rozměrné pole, kde každá buňka bude inicializována na 0.
5. Pro každý pixel v nové bitmapě budeme vypočítávat hodnotu průměru z několika sousedních pixelů v původní (zvětšené) bitmapě (počet sousedních pixelů závisí na hodnotě super-sampling faktoru – počet odpovídá druhé mocnině super-sampling faktoru). Iterujeme tedy dvěma cykly přes celou bitmapu (řádky a sloupce) a pro každý pixel spočítáme průměr hodnot z více pixelů v původní (zvětšené) bitmapě takto
   (a) Budeme iterovat dvěma cykly (na ose X a na ose Y) od 0 do velikosti *super-sampling faktoru*. Aktuální hodnotu iterací (říkejme jim třeba offsetX a offsetY) použijeme pro výpočet souřadnic sousedního pixelu ve zvětšené bitmapě – tyto souřadnice spočítáme vynásobením souřadnice x v nové bitmapě součtem hodnot *super-sampling faktoru* a offsetu na ose X a obdobně pro souřadnici y na ose Y. Výpočet jednotlivých souřadnic sousedních bodů tedy spočítáme podle vztahu:

[1] Násobek pro zvýšení rozlišení, viz bod 1 v části „Postup se super-samplingem“

$$\text{sousedníX} = x \times \text{superSamplingFaktor} + \text{offsetX}$$

$$\text{sousedníY} = y \times \text{superSamplingFaktor} + \text{offsetY}$$

(b) Hodnotu pixelu na vypočítaných souřadnicích přičteme do součtu hodnot pro výpočet aritmetického průměru. Pokud jsme si nevytvořili původní (zvětšenou) bitmapu sami nebo jsme neměli jistotu o správnosti její velikosti, tak je vhodné zkontrolovat, zda jsou souřadnice v rámci původní (zvětšené) bitmapy – to může nastat u krajních pixelů v případě, kdy rozměry původní (zvětšené) bitmapy nejsou přesně dělitelné *super-sampling faktorem*.

(c) Součet hodnot pro výpočet aritmetického průměru vydělíme počtem sousedních pixelů – druhou mocninou *super-sampling faktoru*.

6. Novou bitmapu s aplikovaným anti-aliasingem vypíšeme (pro výpis bitmapy můžeme použít funkci *printBitmap*, viz Zdrojový kód 25).

5.6.8 Kontrolní judge

Judge pro úlohu „Kreslení čáry s anti-aliasingem“ spustíme ve složce `students` pomocí:

```
kotlin judges.main.kts 5_6 data_z_generatoru vystup_z_programu_s_resenim
```

Kapitola 6

Vzorová řešení úloh

V této kapitole najdeme vzorová řešení pro všechny úlohy uvedené v kapitola 5. Vzorová řešení jsou psaná v jazyce Kotlin.

6.1 Vodorovná a svislá čára

Vzorové řešení pro úlohu „Vodorovná a svislá čára“ (zadání viz sekce 5.1) je k dispozici jako Zdrojový kód 5.

```
1  package solution_5_1_functions
2
3  import bitmap.*
4
5  fun drawHorizontalLine(bitmap: Array<Array<Int>>, a_x: Int, b_x: Int, y: Int): Array<Array<Int>> {
6          // proměnná min obsahuje x souřadnici bodu, který je víc vlevo
7          val min_x = minOf(a_x, b_x)
8          // proměnná max obsahuje x souřadnici bodu, který je víc vpravo
9          val max_x = maxOf(a_x, b_x)
10
11         // úprava barvy pixelů na souřadnicích čáry z levého (menšího) bodu pravého (většího) bodu
12         for (i in min_x..max_x) {
13                 bitmap[i][y] = 255
14         }
15
16         return bitmap
17 }
18
19 fun drawVerticalLine(bitmap: Array<Array<Int>>, x: Int, a_y: Int, b_y: Int): Array<Array<Int>> {
20         // proměnná min obsahuje y souřadnici bodu, který je níže
21         val min_y = minOf(a_y, b_y)
22         // proměnná max obsahuje y souřadnici bodu, který je výše
23         val max_y = maxOf(a_y, b_y)
24
25         // úprava barvy pixelů na souřadnicích čáry z dolního (menšího) bodu horního (většího) bodu
26         for (i in min_y..max_y) {
27                 bitmap[x][i] = 255
28         }
29
30         return bitmap
31 }
```

```
32
33  fun solve51Functions(inputFile: String): String {
34
35          val linesList = readInputIntoLinesList(inputFile)
36
37          val (width, height) = linesList[0].split(" ").map { it.toInt() }
38
39          val pointA = linesList[1].toPoint()
40          val pointB = linesList[2].toPoint()
41
42          var bitmap = Array(width) { Array<Int>(height) { 0 } }
43
44          if (pointA.y == pointB.y) {
45                  bitmap = drawHorizontalLine(bitmap, pointA.x, pointB.x, pointA.y)
46          } else if (pointA.x == pointB.x) {
47                  bitmap = drawVerticalLine(bitmap, pointA.x, pointA.y, pointB.y)
48          } else {
49                  return "Vykreslování šikmých čar nejsou podporovány!"
50          }
51
52          return bitmapToString(bitmap, ColorModel.GREY)
53  }
```

Zdrojový kód 5: Vzorové řešení úlohy „Vodorovná a svislá čára“.

6.2 Vodorovná a svislá čára – alternativní řešení

Vzorové řešení pro úlohu „Vodorovná a svislá čára – alternativní řešení“ (zadání viz sekce 5.1) je k dispozici jako Zdrojový kód 6.

```
1   package solution_5_1_alternative
2
3   import bitmap.*
4
5   fun drawAxisLine(a_x: Int, a_y: Int, b_x: Int, b_y: Int, bitmap: Array<Array<Int>>, color: Int =
    ↪  255): Array<Array<Int>> {
6           val rangeX = minOf(a_x, b_x)..maxOf(a_x, b_x)
7           val rangeY = minOf(a_y, b_y)..maxOf(a_y, b_y)
8
9           for (x in rangeX) {
10                  for (y in rangeY) {
11                          bitmap[x][y] = color
12                  }
13          }
14
15          return bitmap
16  }
17
18  fun solve51Alternative(inputFile: String): String {
19      val linesList = readInputIntoLinesList(inputFile)
20
21      val (width, height) = linesList[0].split(" ").map { it.toInt() }
22
23          val pointA = linesList[1].toPoint()
24          val pointB = linesList[2].toPoint()
25
```

```
    var bitmap = Array(width) { Array<Int>(height) { 0 } }

    bitmap = drawAxisLine(pointA.x, pointA.y, pointB.x, pointB.y, bitmap)

    return bitmapToString(bitmap, ColorModel.GREY)
}
```

Zdrojový kód 6: Vzorové řešení úlohy „Vodorovná a svislá čára – alternativní řešení“.

6.3 Čára v libovolném směru mezi 2 body

Vzorové řešení pro úlohu „Čára v libovolném směru mezi 2 body“ (zadání viz sekce 5.2) je k dispozici jako Zdrojový kód 7.

```
package solution_5_2

import kotlin.math.abs
import kotlin.math.roundToInt
import bitmap.*

fun drawLine(a_x: Int, a_y: Int, b_x: Int, b_y: Int, bitmap: Array<Array<Int>>, color: Int = 255):
↪  Array<Array<Int>> {
    val distanceX = b_x - a_x
    val distanceY = b_y - a_y

    //přidat nápovědu do textu, že tady musí být max
    val step = maxOf(abs(distanceX), abs(distanceY))

    val diffX: Double = distanceX.toDouble() / step
    val diffY: Double = distanceY.toDouble() / step

    // přidat nápovědu do textu, že tady musí být min - kreslíme z levého dolního bodu
    var x: Double = a_x.toDouble()
    var y: Double = a_y.toDouble()

    for (i in 0..step) {
        bitmap[x.roundToInt()][y.roundToInt()] = color
        x = x + diffX
        y = y + diffY
    }

    return bitmap
}

fun solve52(inputFile: String): String {
    val linesList = readInputIntoLinesList(inputFile)

    val (width, height) = linesList[0].split(" ").map { it.toInt() }

        val pointA = linesList[1].toPoint()
        val pointB = linesList[2].toPoint()

    var bitmap = Array(width) { Array<Int>(height) { 0 } }

    bitmap = drawLine(pointA.x, pointA.y, pointB.x, pointB.y, bitmap)

```

```
42          return bitmapToString(bitmap, ColorModel.GREY)
43  }
```

Zdrojový kód 7: Vzorové řešení úlohy „Čára v libovolném směru mezi 2 body“.

6.4 Obdélník

Vzorové řešení pro úlohu „Obdélník“ (zadání viz sekce 5.3) je k dispozici jako Zdrojový kód 8.

```
1   package solution_5_3
2
3   import solution_5_1_alternative.*
4   import bitmap.*
5
6   fun solve53(inputFile: String): String {
7       val linesList = readInputIntoLinesList(inputFile)
8
9       val (width, height) = linesList[0].split(" ").map { it.toInt() }
10
11      val (a_x, a_y) = linesList[1].split(" ").map { it.toInt() }
12      val (b_x, b_y) = linesList[2].split(" ").map { it.toInt() }
13      val (c_x, c_y) = linesList[3].split(" ").map { it.toInt() }
14      val (d_x, d_y) = linesList[4].split(" ").map { it.toInt() }
15
16
17      var bitmap = Array(width) { Array<Int>(height) { 0 } }
18
19
20      bitmap = drawAxisLine(a_x, a_y, b_x, b_y, bitmap)
21      bitmap = drawAxisLine(b_x, b_y, c_x, c_y, bitmap)
22      bitmap = drawAxisLine(c_x, c_y, d_x, d_y, bitmap)
23      bitmap = drawAxisLine(d_x, d_y, a_x, a_y, bitmap)
24
25
26      return bitmapToString(bitmap, ColorModel.GREY)
27  }
```

Zdrojový kód 8: Vzorové řešení úlohy „Obdélník“.

```
1   package solution_5_1_alternative
2
3   import bitmap.*
4
5   fun drawAxisLine(a_x: Int, a_y: Int, b_x: Int, b_y: Int, bitmap: Array<Array<Int>>, color: Int =
    ↪  255): Array<Array<Int>> {
6           val rangeX = minOf(a_x, b_x)..maxOf(a_x, b_x)
7           val rangeY = minOf(a_y, b_y)..maxOf(a_y, b_y)
8
9           for (x in rangeX) {
10                  for (y in rangeY) {
11                          bitmap[x][y] = color
12                  }
13          }
14
```

```
        return bitmap
}
```

Zdrojový kód 9: Dodatek k řešení úlohy „Obdélník“.

6.5 Vyplněný obdélník

Vzorové řešení pro úlohu „Vyplněný obdélník“ (zadání viz sekce 5.4) je k dispozici jako Zdrojový kód 10.

```
package solution_5_4

import solution_5_1_alternative.*
import bitmap.*

fun solve54(inputFile: String): String {
    val linesList = readInputIntoLinesList(inputFile)

    val (width, height) = linesList[0].split(" ").map { it.toInt() }

        val pointA = linesList[1].toPoint()
        val pointB = linesList[2].toPoint()

    var bitmap = Array(width) { Array<Int>(height) { 0 } }


    bitmap = drawAxisLine(pointA.x, pointA.y, pointB.x, pointB.y, bitmap)

    return bitmapToString(bitmap, ColorModel.GREY)
}
```

Zdrojový kód 10: Vzorové řešení úlohy „Vyplněný obdélník“.

```
package solution_5_1_alternative

import bitmap.*

fun drawAxisLine(a_x: Int, a_y: Int, b_x: Int, b_y: Int, bitmap: Array<Array<Int>>, color: Int =
↪  255): Array<Array<Int>> {
        val rangeX = minOf(a_x, b_x)..maxOf(a_x, b_x)
        val rangeY = minOf(a_y, b_y)..maxOf(a_y, b_y)

        for (x in rangeX) {
                for (y in rangeY) {
                        bitmap[x][y] = color
                }
        }

        return bitmap
}
```

Zdrojový kód 11: Dodatek k řešení úlohy „Vyplněný obdélník“.

6.6 Mnohoúhelník

Vzorové řešení pro úlohu „Mnohoúhelník“ (zadání viz sekce 5.5) je k dispozici jako Zdrojový kód 12.

```kotlin
package solution_5_5

import solution_5_2.*
import bitmap.*
import kotlin.math.abs
import kotlin.math.roundToInt

fun solve55(inputFile: String): String {
    val linesList = readInputIntoLinesList(inputFile)

    val (width, height) = linesList[0].split(" ").map { it.toInt() }

    val pointsSize = linesList[1].toInt()

    val points = mutableListOf<Point>()

    for (i in 0 until pointsSize) {
        points.add(linesList[i + 2].toPoint())
        //val (x, y) = linesList[i + 2].split(" ").map { it.toInt() }
        //points.add(Pair(x, y))
    }

    var bitmap = Array(width) { Array<Int>(height) { 0 } }

    for (i in 0 until pointsSize - 1) {
        bitmap = drawLine(points[i].x, points[i].y, points[i + 1].x, points[i + 1].y, bitmap)
    }

    bitmap = drawLine(points[pointsSize - 1].x, points[pointsSize - 1].y, points[0].x, points[0].y,
    ↪ bitmap)


    return bitmapToString(bitmap, ColorModel.GREY)
}

```

Zdrojový kód 12: Vzorové řešení úlohy „Mnohoúhelník“.

```kotlin
package solution_5_2

import kotlin.math.abs
import kotlin.math.roundToInt
import bitmap.*

fun drawLine(a_x: Int, a_y: Int, b_x: Int, b_y: Int, bitmap: Array<Array<Int>>, color: Int = 255):
↪ Array<Array<Int>> {
    val distanceX = b_x - a_x
    val distanceY = b_y - a_y

    //přidat nápovědu do textu, že tady musí být max
    val step = maxOf(abs(distanceX), abs(distanceY))

    val diffX: Double = distanceX.toDouble() / step
```

```
15      val diffY: Double = distanceY.toDouble() / step
16
17      // přidat nápovědu do textu, že tady musí být min - kreslíme z levého dolního bodu
18      var x: Double = a_x.toDouble()
19      var y: Double = a_y.toDouble()
20
21      for (i in 0..step) {
22          bitmap[x.roundToInt()][y.roundToInt()] = color
23          x = x + diffX
24          y = y + diffY
25      }
26
27      return bitmap
28  }
29
30  fun solve52(inputFile: String): String {
```

Zdrojový kód 13: Dodatek k řešení úlohy „Mnohoúhelník“.

6.6.1 Generátor pro úlohu „Mnohoúhelník“

U této úlohy může být také zajímavé se zamyslet nad tím, jak vlastně funguje generátor vstupních dat, respektive generování jednotlivých bodů. Nejdříve vygenerujeme počet bodů n, kde n je větší než 3 a zároveň $360 \mod n = 0$ (jinými slovy lze kružnici rozdělit na n stejně velkých částí). Následně vygenerujeme náhodný průměr kružnice, který velikostně zapadá do bitmapy. V závěru již pouze vypočítáme souřadnice bodů na této kružnici.

Při výpočtu a následném vykreslování pravidelného mnohoúhelníku je však třeba vzít v úvahu omezení, dané použitím celočíselných souřadnic. Každý vrchol mnohoúhelníku je vypočítán na základě souřadnic odpovídajících bodům na kružnici, ale zaokrouhlování může způsobit drobné odchylky od ideální polohy vrcholů. Tyto odchylky mohou vést k tomu, že výsledný mnohoúhelník nebude zcela pravidelný, zejména v případě menšího počtu stran.

S rostoucím počtem stran n se však tyto nepřesnosti stávají méně patrnými, protože rozdíly způsobené zaokrouhlováním jsou při vyšším n relativně malé a na vykresleném obrázku lidským okem téměř nerozeznatelné. V rámci daného rozlišení a zvolené velikosti mnohoúhelníku tedy výsledná iluze kružnice zůstává zachována, a to i při použití celočíselných souřadnic.

6.7 Kreslení čáry s anti-aliasingem

Vzorové řešení pro úlohu „Kreslení čáry s anti-aliasingem“ (zadání viz podsekce 5.6.2) je k dispozici jako Zdrojový kód 14.

```
1  package solution_5_6
2
3  import solution_5_2.*
4  import bitmap.*
5  import kotlin.math.abs
6  import kotlin.math.roundToInt
```

```
7
8   fun applySupersamplingAntialiasing(bitmap: Array<Array<Int>>, supersampleFactor: Int):
    ↪  Array<Array<Int>> {
9       val originalHeight = bitmap.size
10      val originalWidth = bitmap[0].size
11
12      val newHeight = originalHeight / supersampleFactor
13      val newWidth = originalWidth / supersampleFactor
14
15      // nová bitmapa do které se aplikuje anti-aliasing
16      val result = Array(newHeight) { Array(newWidth) { 0 } }
17
18      for (y in 0 until newHeight) {
19          for (x in 0 until newWidth) {
20              var sum = 0
21              for (pixelOffsetY in 0 until supersampleFactor) {
22                  for (pixelOffsetX in 0 until supersampleFactor) {
23                      val pixelY = y * supersampleFactor + pixelOffsetY
24                      val pixelX = x * supersampleFactor + pixelOffsetX
25                      // if (pixelY < originalHeight && pixelX < originalWidth) // případná kontrola,
                        ↪  viz krok 5b
26                      sum += bitmap[pixelY][pixelX]
27                  }
28              }
29              result[y][x] = sum / (supersampleFactor * supersampleFactor)
30          }
31      }
32
33      return result
34  }
35
36  fun solve56(inputFile: String): String {
37      val linesList = readInputIntoLinesList(inputFile)
38
39
40      val (width, height) = linesList[0].split(" ").map { it.toInt() }
41
42      val superSamplingFactor = linesList[1].toInt()
43
44      val pointPairCount = linesList[2].toInt()
45
46      val pointPairs = mutableListOf<Pair<Point, Point>>()
47
48
49      for (i in 0 until pointPairCount) {
50          val (firstPoint, secondPoint) = linesList[i + 3].split(",").map { it.toPoint() *
            ↪  superSamplingFactor }
51          pointPairs.add(Pair(firstPoint, secondPoint))
52      }
53
54      // původní zvětšená bitmapa
55      var bitmap = Array(width * superSamplingFactor) { Array<Int>(height * superSamplingFactor) { 0 }
        ↪  }
56
57      for (i in 0 until pointPairCount) {
58          bitmap = drawLine(pointPairs[i].first.x, pointPairs[i].first.y, pointPairs[i].second.x,
            ↪  pointPairs[i].second.y, bitmap)
59      }
60
61      val outputBitmap = applySupersamplingAntialiasing(bitmap, superSamplingFactor)
```

```
62
63      return bitmapToString(outputBitmap, ColorModel.GREY)
64  }
65
```

Zdrojový kód 14: Vzorové řešení úlohy „Kreslení čáry s anti-aliasingem“.

```
1   package solution_5_2
2
3   import kotlin.math.abs
4   import kotlin.math.roundToInt
5   import bitmap.*
6
7   fun drawLine(a_x: Int, a_y: Int, b_x: Int, b_y: Int, bitmap: Array<Array<Int>>, color: Int = 255):
    ↪  Array<Array<Int>> {
8       val distanceX = b_x - a_x
9       val distanceY = b_y - a_y
10
11      //přidat nápovědu do textu, že tady musí být max
12      val step = maxOf(abs(distanceX), abs(distanceY))
13
14      val diffX: Double = distanceX.toDouble() / step
15      val diffY: Double = distanceY.toDouble() / step
16
17      // přidat nápovědu do textu, že tady musí být min - kreslíme z levého dolního bodu
18      var x: Double = a_x.toDouble()
19      var y: Double = a_y.toDouble()
20
21      for (i in 0..step) {
22          bitmap[x.roundToInt()][y.roundToInt()] = color
23          x = x + diffX
24          y = y + diffY
25      }
26
27      return bitmap
28  }
29
30  fun solve52(inputFile: String): String {
```

Zdrojový kód 15: Dodatek k řešení úlohy „Kreslení čáry s anti-aliasingem“.

Kapitola 7

Knihovna s pomocnými funkcemi

V této kapitole si představíme knihovnu funkcí, která byla navržena tak, aby usnadnila programování opakujících se nebo složitějších částí úloh. Její hlavní funkcí je automatizovat úkoly, které by jinak mohly být časově náročné nebo technicky složité, čímž uvolňuje studentům čas a energii na soustředění se na samotné základy úlohy. Cílem je, vytvořit prostředí umožňující studentům efektivněji se učit a soustředit se na klíčové koncepty, místo aby se zbytečně zdržovali implementací složitých detailů. Zdrojové kódy knihovny se nacházejí v příloze ve složce *students/lib_sources/*.

7.1 Popis jednotlivých částí

7.1.1 Objekt ColorModel

ColorModel je v zásadě jen množina několika barevných modelů, na které se můžeme dle potřeby odkazovat, například pomocí `ColorModel.RGB`.

```
27 enum class ColorModel(val modelName: String) {
28     BW("BW"),
29     GREY("GREY"),
30     RGB("RGB"),
31     CMYK("CMYK"),
32     HSV("HSV"),
33     LAB("LAB"),
34 }
```

Zdrojový kód 16: Objekt ColorModel

7.1.2 Objekt Point

Point je objekt s vlastnostmi x a y, které slouží pro ukládání souřadnic bodů. Objekt má podporu pro převod na řetězec (ve formátu pro naše potřeby) a má přetížený operátor pro násobení – při násobení bodu se jeho x a y vynásobí daným činitelem.

```
36 data class Point(val x: Int, val y: Int) {
37     // Převede bod do textové reperezentace.
38     override fun toString(): String = "$x $y"
39 
40     // Přetížení operátoru násobení
41     // Bod zvětší xkrát. Resp. jednotlivé souřadnice bodu jsou vynásobeny koeficientem scale.
42     operator fun times(scale: Int): Point = Point(x * scale, y * scale)
43 }
```

Zdrojový kód 17: Objekt Point

Převod řetězce na bod

Knihovna také přidává rozšíření typu *String* o funkce *toPoint* a *toPoints*, které nám pomáhají převádět řetězce na objekt `Point`.

```
47 fun String.toPoint(): Point {
48     val (x, y) = this.trim().split(" ").map { it.toIntOrNull() ?: error("Invalid number format in
       ↪  '$this'") }
49     return Point(x, y)
50 }
51 
52 // Rozšíření String o parsování na list Point
53 // Převede řetězec obsahující seznam bodů oddělených čárkami na list Point.
54 fun String.toPoints(): List<Point> {
55     return this.split(",").map { it.toPoint() }
56 }
```

Zdrojový kód 18: String.toPoint a String.toPoints

7.1.3 Funkce random

Funkce *random* přijímá dva číselné parametry *start* a *end*. Vrací vygenerované náhodné číslo mezi start (včetně) a end (kromě).

```
13 fun random(start: Int, end: Int): Int {
14     return Random.nextInt(start, end)
15 }
```

Zdrojový kód 19: Funkce random

7.1.4 Funkce randomDistinct

Funkce *randomDistinct* přijímá tři číselné parametry *start*, *end* a *count*. Vrací seznam o délce *count* s náhodně vygenerovanými čísly mezi start (včetně) a end (kromě).

```
18 fun randomDistinct(start: Int, end: Int, count: Int): List<Int> {
19     return generateSequence { Random.nextInt(start, end) }.distinct().take(count).toList()
20 }
```

Zdrojový kód 20: Funkce randomDistinct

7.1.5 Funkce randomBoolean

Funkce *randomBoolean* vrací náhodnou hodnotu typu *Boolean* - tedy buď `true` nebo `false`.

```
23 fun randomBoolean(): Boolean {
24     return Random.nextBoolean()
25 }
```

Zdrojový kód 21: Funkce randomBoolean

7.1.6 Funkce generateDistinctPoints

Funkce *generateDistinctPoints* přijímá dva parametry *rangeX* a *rangeY* typu IntRange a jeden číselný parametr *count*.

Vrací seznam objektů Point (bod) o délce *count*. Jednotlivé body mají náhodně vygenerované souřadnice v rámci *rangeX* a *rangeY*, všechny body jsou vzájemně různé.

Podobně fungují také funkce *generateDistinctVerticalPoints* a *generateDistinctHorizontalPoints* s tím rozdílem, že body jsou různé pouze na ose y, respektive na ose x.

```kotlin
fun generateDistinctPoints(rangeX: IntRange, rangeY: IntRange, count: Int): List<Point> {
    require(count >= 0) { "Count must be non-negative, was $count" }
    val totalPoints = (rangeX.last - rangeX.first) * (rangeY.last - rangeY.first)
    require(totalPoints >= count) {
        "Range does not contain enough unique points for the requested count. Available: $totalPoints,
        ↪  requested: $count"
    }


    return generateSequence { Point(Random.nextInt(rangeX.start, rangeX.last),
    ↪  Random.nextInt(rangeY.first, rangeY.last)) }
        .distinct()
        .take(count)
        .toList()
}

// Vygeneruje seznam (o délce count) vzájemně odlišných bodů (Point) na stejné vodorovné přímce v
↪  zadaném rozsahu souřadnic
fun generateDistinctHorizontalPoints(rangeX: IntRange, rangeY: IntRange, count: Int): List<Point> {
    require(count >= 0) { "Count must be non-negative, was $count" }
    require(rangeY.last - rangeY.start + 1 >= count) { "RangeY must be at least as large as count.
    ↪  Count was $count." }

    val y = Random.nextInt(rangeY.first, rangeY.last + 1)

    return generateSequence { Point(Random.nextInt(rangeX.start, rangeX.last + 1), y) }
        .distinct()
        .take(count)
        .toList()
}

// Vygeneruje seznam (o délce count) vzájemně odlišných bodů (Point) na stejné svislé přímce v zadaném
↪  rozsahu souřadnic
fun generateDistinctVerticalPoints(rangeX: IntRange, rangeY: IntRange, count: Int): List<Point> {
    require(count >= 0) { "Count must be non-negative, was $count" }
    require(rangeX.last - rangeX.start + 1 >= count) { "RangeX must be at least as large as count.
    ↪  Count was $count." }

    val x = Random.nextInt(rangeX.first, rangeX.last + 1)

    return generateSequence { Point(x, Random.nextInt(rangeY.start, rangeY.last + 1)) }
        .distinct()
        .take(count)
        .toList()
}
```

Zdrojový kód 22: Funkce generateDistinctPoints, generateDistinctHorizontalPoints a generateDistinctVerticalPoints

7.1.7 Funkce readInputIntoLinesList

Funkce *readInputIntoLinesList* přijímá v parametrech jeden řetězec *filename*.

Funkce zkontroluje, zda soubor s názvem *filename* existuje a pokud ano, tak vrací seznam řetězců se všemi řádky tohoto souboru.

```
100 fun readInputIntoLinesList(filename: String): List<String> {
101     val file = Path(filename)
102     require(file.exists()) { "File '$filename' does not exist." }
103     val linesList = file.readLines()
104 
105     return linesList
106 }
```

Zdrojový kód 23: Funkce readInputIntoLinesList

7.1.8 Funkce bitmapToString

Funkce *bitmapToString* přijímá dva parametry. Parametr *bitmap* typu *Array<Array <Int> >* a parametr *colorModel* typu *ColorModel*.

Funkce tuto bitmapu převede do řetězcové podoby tak, jak ji specifikujeme v 1. kapitole. Na základě parametru *colorModel* obsahuje hlavička bitmapy informaci o zvoleném barevném modelu. Řetezec s bitmapou funkce vrátí zpět.

```
109 fun bitmapToString(bitmap: Array<Array<Int>>, colorModel: ColorModel): String {
110     val width = bitmap.size
111     val height = bitmap[0].size
112 
113     require(bitmap.all { it.size == height }) { "All rows must have the same pixel count." }
114 
115     val rows = (height - 1 downTo 0).map { y ->
116         (0 until width).joinToString(" ") { x ->
117             bitmap[x][y].toString()
118         }
119     }
120 
121     return buildString {
122         appendLine("BMP,$colorModel")
123         appendLine("$width $height")
124         append(rows.joinToString("\n"))
125     }
126 }
```

Zdrojový kód 24: Funkce bitmapToString

7.1.9 Funkce printBitmap

Funkce *printBitmap* přijímá dva parametry. Parametr *bitmap* typu *Array<Array <Int> >* a parametr *colorModel* typu *ColorModel*.

S těmito parametry zavolá funkci `bitmapToString` a vrácený řetězec vypíše do konzole.

```
129 fun printBitmap(bitmap: Array<Array<Int>>, colorModel: ColorModel = ColorModel.GREY) {
130     println(bitmapToString(bitmap, colorModel))
131 }
```

Zdrojový kód 25: Funkce printBitmap

7.1.10 Funkce parseBitmapHeader

Funkce *parseBitmapHeader* přijímá v parametrech jeden řetězec *header*.

Funkce zkontroluje, zda je hlavička v řetězci *header* validní a pokud ano, tak funkce vrátí barevný model, který tato hlavička uvádí. Vrací objekt typu *ColorModel*.

```
134 fun parseBitmapHeader(header: String): ColorModel {
135     val headerParts = header.split(",")
136     require(headerParts.size == 2) { "Invalid header format." }
137 
138     val colorModel = ColorModel.values().find { it.modelName == headerParts[1] }
139     require(colorModel != null) { "Invalid color model." }
140 
141     return colorModel
142 }
```

Zdrojový kód 26: Funkce parseBitmapHeader

7.1.11 Funkce stringToBitmap

Funkce *stringToBitmap* přijímá v parametrech jeden řetězec *input*.

Funkce zpracuje hlavičku a na zpracování dat samotného obrázku zavolá funkci `parseBitmapImageData` a poté vrátí její výstup.

```
145 fun stringToBitmap(input: String): Array<Array<Int>> {
146     val linesList = input.lines()
147 
148     val header = linesList[0].split(",")
149     val (width, height) = linesList[1].split(" ").map { it.toInt() }
150 
151     return parseBitmapImageData(width, height, linesList.drop(2))
152 }
```

Zdrojový kód 27: Funkce stringToBitmap

7.1.12 Funkce parseBitmapImageData

Funkce *parseBitmapImageData* přijímá dva číselné parametry *width* a *height*. Jako třetí parametr funkce přijímá seznam řetězců *linesList*.

Funkce zkontroluje, zda řetezce obsahují správný počet pixelů a následně je převede do 2rozměrného pole. Funkce vrací 2rozměrné pole s převedenými pixely.

```
155 fun parseBitmapImageData(width: Int, height: Int, linesList: List<String>): Array<Array<Int>> {
156
157     val bitmap = Array(width) { Array<Int>(height) { 0 } }
158
159     for (y in 0 until height) {
160         val row = linesList[y].split(" ")
161         for (x in 0 until width) {
162             bitmap[x][height - y - 1] = row[x].toInt()
163         }
164     }
165
166     return bitmap
167 }
```

Zdrojový kód 28: Funkce parseBitmapImageData

7.1.13 Funkce saveRGBPixel

Funkce *saveRGBPixel* přijímá jeden parametr *color* typu *Color*. Pomocí techniky bit packing funkce uloží tři 8bitové barevné složky do jednoho 32bitového čísla - to je možné díky tzv. technice bit packingu[34]. Můžeme si například definovat, že bity 0 - 7 obsahují modrou složku, bity 8 až 15 zelenou složku a bity 16 až 23 červenou složku. Pokud barvu podle těchto pravidel zakódujeme do čísla, tak podle stejných pravidel jsme schopni zpětně z daného čísla spočítat hodnoty jednotlivých barevných složek.

Funkce vrací číslo se zakódovanými všemi třemi složkami.

```
174 fun saveRGBPixel(color: Color): Int {
175     var pixelValue = color.getRed()
176     pixelValue = pixelValue.shl(8) + color.getGreen()
177     pixelValue = pixelValue.shl(8) + color.getBlue()
178     return pixelValue
179 }
```

Zdrojový kód 29: Funkce saveRGBPixel

7.1.14 Funkce parseRGBPixel

Funkce *parseRGBPixel* pracuje na podobném principu jako funkce `saveRGBPixel` jen s tím rozdílem, že místo barvy v parametrech přijímá funkce číslo. Toto číslo v sobě obsahuje 3 barevné složky R, G a B. Po dekódování funkce vrátí danou barvu v podobě objektu *Color*.

```
182 fun parseRGBPixel(pixelValue: Int): Color {
183     val red = (pixelValue shr 16) and 0xFF
184     val green = (pixelValue shr 8) and 0xFF
185     val blue = pixelValue and 0xFF
186     return Color(red, green, blue)
187 }
```

Zdrojový kód 30: Funkce parseRGBPixel

7.1.15 Funkce saveBitmapToPngFile

Funkce *saveBitmapToPngFile* přijímá tři parametry. Parametr *bitmap* typu *Array< Array <Int> >*, v druhém parametru řetězec *filename* a třetí parametr *colorModel* typu *ColorModel*.
Funkce uloží bitmapu do obrázku ve formátu PNG. Funkce aktuálně podporuje pouze barevný model Grey, pro další barevné modely je možné funkci rozšířit.

```
190 fun saveBitmapToPngFile(bitmap: Array<Array<Int>>, filename: String, colorModel: ColorModel =
    ↪ ColorModel.GREY) {
191     val width = bitmap.size
192     val height = bitmap[0].size
193 
194     require(bitmap.all { it.size == height }) { "All rows must have same pixel count." }
195 
196     val image = BufferedImage(width, height, BufferedImage.TYPE_INT_RGB) // nastavení barevného modelu,
        ↪ který budeme používat při ukládání
197 
198     for (y in 0 until height) {
199         for (x in 0 until width) {
200             // pro další barevné modely je nutné rozšířit tuto část pro zpracování barvy pixelu
201             val pixelValue = bitmap[x][height - y - 1] // obrácený index pro řádky
202             val color = Color(pixelValue, pixelValue, pixelValue)
203             image.setRGB(x, y, color.rgb)
204         }
205     }
206 
207     val path = Path(filename)
208     path.toFile().also { file ->
209         ImageIO.write(image, "png", file)
210     }
211 }
```

Zdrojový kód 31: Funkce saveBitmapToPngFile

Kapitola 8

Závěr

8.1 Shrnutí práce

Tato bakalářská práce si kladla za cíl vytvořit vzdělávací materiály pro výuku programování s využitím počítačové grafiky. Výsledkem je sada úloh, která kombinuje teoretické základy s praktickými aktivitami, podporujícími algoritmické myšlení a vizualizaci výsledků. Hlavním přínosem práce je vytvoření ucelených výukových materiálů, přizpůsobených studentům s různými úrovněmi dovedností díky gradovaným úlohám.

Teoretická část práce se zaměřila na klíčové koncepty z oblasti počítačové grafiky, jako jsou rozdíly mezi rastrovou a vektorovou grafikou, barevné modely (RGB, HSV, LAB aj.), specifikace formátu BMP a principy anti-aliasingu. Tyto teoretické základy poskytují studentům nezbytné informace pro pochopení a řešení praktických úloh. Teoretická část zároveň slouží jako zdroj podkladů pro učitele, kteří mohou informace dále přizpůsobit konkrétním potřebám výuky.

Praktická část práce obsahuje sadu úloh zaměřených na vykreslování grafických objektů, jako jsou čáry, mnohoúhelníky a vyplněné plochy, zahrnuje také implementaci anti-aliasingu. Tyto úlohy využívají principy open-datových úloh, které studentům poskytují flexibilitu při výběru programovacího jazyka a větší prostor pro samostatnou práci. Práce rovněž obsahuje generátory zadání a podpůrnou knihovnu funkcí, usnadňujících řešení úloh a poskytujících okamžitou zpětnou vazbu prostřednictvím kontrolního programu (judge). V této kombinaci se tak jedná o malý ekosystém, který si mohou vyučující sami dle svých individuálních potřeb rozšiřovat o další úlohy a dlouhodobě jej pravidelně používat při výuce. Materiály navíc využívají prvky gamifikace a vizualizace, které zvyšují motivaci studentů a usnadňují pochopení klíčových konceptů.

Celkově tato práce ukazuje, že propojení programování a počítačové grafiky je nejen atraktivní, ale i efektivní způsob, jak přiblížit studentům klíčové principy programování, rozvíjet jejich schopnosti a připravit je na reálné výzvy v IT praxi.

8.2 Možná rozšíření

Do budoucna by materiály mohly být rozšířeny o pokročilé grafické algoritmy, jako je vykreslování Bézierových křivek, algoritmy pro výplň oblasti či práci se stíny. Dalším směrem rozvoje by mohlo být

zaměření na 2D grafiku, například animace objektů a práce s texturami, a přechod k základům 3D grafiky. Úlohy na téma vykreslování trojúhelníků, jednoduché transformace a práce s perspektivou by mohly studentům přiblížit moderní principy používané v oblasti počítačové grafiky.

Bibliografie

[1] Pavel Töpfer a Karel Horák. *Algoritmy a programovací techniky*. cze. Prometheus, Praha, 1. vyd. Vydání, 1995. ISBN: 978-80-85849-83-7. OCLC: 36614248.

[2] Martin Mareš a Tomáš Valla. *Průvodce labyrintem algoritmů*. cze, číslo 15. publikace in CZ.NIC. CZ.NIC, Praha, 1. vydání. Vydání, 2017. ISBN: 978-80-88168-19-5. OCLC: 1000492240.

[3] Gomez Graphics Vector Conversions. Raster (Bitmap) vs Vector, listopad 2024. URL: https://vector-conversions.com/vectorizing/raster_vs_vector.html (citováno 23. 11. 2024).

[4] George H. Joblove a Donald Greenberg. Color spaces for computer graphics. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '78, strany 20–25, New York, NY, USA. Association for Computing Machinery, 1978. ISBN: 978-1-4503-7908-3. DOI: 10.1145/800248.807362. URL: https://dl.acm.org/doi/10.1145/800248.807362 (citováno 23. 11. 2024).

[5] Dan Margulis. *Photoshop LAB color : the canyon conundrum and other adventures in the most powerful colorspace*. eng. Berkeley, Calif. : Peachpit Press, 2006. ISBN: 978-0-321-35678-9. URL: http://archive.org/details/photoshoplabcolo0000marg (citováno 23. 11. 2024).

[6] Karsten Lehn, Merijam Gotzes a Frank Klawonn. Greyscale and Colour Representation. en. In Karsten Lehn, Merijam Gotzes a Frank Klawonn, editoři, *Introduction to Computer Graphics: Using OpenGL and Java*, strany 193–210. Springer International Publishing, Cham, 2023. ISBN: 978-3-031-28135-8. DOI: 10.1007/978-3-031-28135-8_6. URL: https://doi.org/10.1007/978-3-031-28135-8_6 (citováno 23. 11. 2024).

[7] James D. Murray a William VanRyper, editoři. *Encyclopedia of graphics file formats: the complete reference on CD-ROM with links to Internet resources; for PC, Macintosh and UNIX platforms. Buch*. O'Reilly, Bonn, 2. ed vydání, 1996. ISBN: 978-1-56592-161-0.

[8] Gsliepen. Naa-vs-aa.png (PNG Image, 640 × 1352 pixels) — Scaled (69%), březen 2016. URL: https://commons.wikimedia.org/wiki/File:Naa-vs-aa.png (citováno 23. 11. 2024).

[9] Silvia Weinzettelova. Traditional Type in the Digital Era. *Bulletin-Prague College Centre for Research and Interdisciplinary Studies*:5–24, 2012.

[10] Organizátoři KSP. Pravidla KSP-H, listopad 2024. URL: https://ksp.mff.cuni.cz/h/pravidla/#prakticke-ulohy (citováno 24. 11. 2024).

[11] Eric Wastl. Advent of Code 2024, 2024. URL: https://adventofcode.com/ (citováno 28. 11. 2024).

[12] Organizátoři Kasiopea. Kasiopea - vstup a výstup, 2024. URL: https://kasiopea.matfyz.cz/tipytriky/vstup_vystup/ (citováno 28. 11. 2024).

[13] Organizátoři KSP. Korespondenční seminář z programování MFF UK, 2024. URL: https://ksp.mff.cuni.cz/ (citováno 28. 11. 2024).

[14] Digitální a informační agentura. Datové sady - Národní katalog otevřených dat (NKOD), 2024. URL: https://data.gov.cz/datov%C3%A9-sady (citováno 28. 11. 2024).

[15] Thomas W. Malone. Toward a theory of intrinsically motivating instruction. *Cognitive Science*, 5(4):333–369, 1981. ISSN: 0364-0213. DOI: https://doi.org/10.1016/S0364-0213(81)80017-1. URL: https://www.sciencedirect.com/science/article/pii/S0364021381800171.

[16] JetBrains. Kotlin Programming Language - Why Teach. en, 2024. URL: https://kotlinlang.org/education/why-teach-kotlin.html (citováno 28. 11. 2024).

[17] fwcd. Kotlin - Visual Studio Marketplace. en-us, červenec 2024. URL: https://marketplace.visualstudio.com/items?itemName=fwcd.kotlin (citováno 26. 11. 2024).

[18] JetBrains. Releases · JetBrains/kotlin. en, listopad 2024. URL: https://github.com/JetBrains/kotlin/releases (citováno 26. 11. 2024).

[19] SDKMAN! Home | SDKMAN! the Software Development Kit Manager. en, listopad 2024. URL: https://sdkman.io/ (citováno 26. 11. 2024).

[20] mattwojo. Install WSL. en-us, listopad 2024. URL: https://learn.microsoft.com/en-us/windows/wsl/install (citováno 26. 11. 2024).

[21] Chocolatey Software, Inc. Installing Chocolatey. en, listopad 2024. URL: https://chocolatey.org/install (citováno 26. 11. 2024).

[22] Chocolatey Software, Inc. Chocolatey - Kotlin Compiler Install. en, červen 2021. URL: https://community.chocolatey.org/packages/kotlinc (citováno 26. 11. 2024).

[23] JetBrains. kotlin-stdlib. en-US, 2024. URL: https://kotlinlang.org/api/core/kotlin-stdlib/ (citováno 28. 11. 2024).

[24] Jetbrains. Kotlin Docs | Kotlin. en-US, 2024. URL: https://kotlinlang.org/docs/home.html (citováno 28. 11. 2024).

[25] Stylianos Gakis. Can't get fleet smart mode to autoformat my code, nor provide any autocompletion options : FL-22261, 2023. URL: https://youtrack.jetbrains.com/issue/FL-22261/Cant-get-fleet-smart-mode-to-autoformat-my-code-nor-provide-any-autocompletion-options (citováno 30. 11. 2024).

[26] ORYLY. Scripts: @file:Import() in kotlin-main-kts uses a stale cache : KT-42101, 2020. URL: https://youtrack.jetbrains.com/issue/KT-42101/Scripts-fileImport-in-kotlin-main-kts-uses-a-stale-cache (citováno 28. 11. 2024).

[27] Organizátoři Kasiopea. Vstup a výstup – Kasiopea, 2024. URL: https://kasiopea.matfyz.cz/tipytriky/vstup_vystup/ (citováno 30. 11. 2024).

[28] Alan H. Watt. *3D computer graphics*. Addison-Wesley, Harlow, England ; Reading, Mass, 3rd ed vydání, 2000. ISBN: 978-0-201-39855-7.

[29] Alois Zingl. A Rasterizing Algorithm for Drawing Curves. en, 2012. URL: http://members.chello.at/~easyfilter/Bresenham.pdf.

[30] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965. ISSN: 0018-8670. DOI: 10.1147/sj.41.0025. URL: https://ieeexplore.ieee.org/document/5388473 (citováno 28. 11. 2024). Conference Name: IBM Systems Journal.

[31] Xiaolin Wu. An efficient antialiasing technique. In *Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '91, strany 143–152, New York, NY, USA. Association for Computing Machinery, 1991. ISBN: 978-0-89791-436-9. DOI: 10.1145/122718.122734. URL: https://dl.acm.org/doi/10.1145/122718.122734 (citováno 28. 11. 2024).

[32] jam-master jim. supersampling - Everything2.com, březen 2004. URL: https://www.everything2.com/index.pl?node_id=1028947 (citováno 28. 11. 2024).

[33] Allen Sherrod. *Game Graphic Programming*. en. Course Technology/Charles River Media/Cengage Learning, 2008. ISBN: 978-1-58450-615-7. Google-Books-ID: gasLAAAAQBAJ.

[34] Maxim Zaks. Smart way of storing data. Let's talk about bit packing... | by Maxim Zaks | Towards Data Science, 2017. URL: https://towardsdatascience.com/smart-way-of-storing-data-d22dd5077340 (citováno 01. 12. 2024).

Seznam ukázek zdrojového kódu

Seznam obrázků

Seznam příloh

Příloha je tvořena komprimovaným souborem zip obsahujícím:

- Složka **generators_sources** obsahuje zdrojové kódy jednotlivých generátorů vstupních dat. Každý generátor je implementován jako samostatný Kotlin soubor. Například *generator_4.kt* obsahuje funkci generující data pro úlohu číslo 4, zatímco *generator_5_1.kt* až *generator_5_6.kt* pokrývají různé varianty úloh z kapitola 5.
- Složka **scripts** obsahuje pomocné shellové skripty pro kompilování zdrojových kódů a skript na otestování základní funkčnosti zdrojových kódů. Konkrétně:
  - **compile_all.sh**: Spustí všechny kompilační skripty.
  - **compile_generator_lib.sh**: Kompiluje knihovnu pro generátory.
  - **compile_judge_lib.sh**: Kompiluje knihovnu pro judge.
  - **test_tasks.sh**: Skript provede několik iterací generování testovacích dat, spuštění řešení a judge. Zkontroluje, zda se vše dokončí bez chyby a zda judge vrátí správný výsledek.
- Složka **solutions** obsahuje řešení úloh a pomocnou knihovnu:
  - **lib.jar** je pomocná knihovna vytvořená z *bitmap.kt*.
  - **solutions.main.kts** je skript zajišťující spuštění jednotlivých řešení.
  - Složka **sources** obsahuje zdrojové kódy konkrétních řešení. Každý soubor odpovídá jedné variantě úlohy, například *solution_4.kt* a *solution_4_hard.kt* pro různé obtížnosti úlohy 4, či *solution_5_1_functions.kt* a *solution_5_1_alternative.kt* jako alternativní přístupy k úloze 5.1.
- Složka **students** je určena pro použití studenty a obsahuje následující položky:
  - Soubory **convert_bitmap_to_png.main.kts, generators.main.kts a judges.main.kts** poskytují studentům nástroje pro práci s konvertorem bitmapy, generátory a judge systémy.
  - Složka **libs** zahrnuje zkompilované knihovny (generators.jar, judges.jar, lib.jar) pro generování vstupů, kontrolu výstupů a sdílenou funkcionalitu.
  - Složka **lib_sources** obsahuje zdrojový kód (*bitmap.kt*) a skript *compile_lib.sh* pro kompilaci vlastní knihovny.